Technical Report
CMU/SEI-88-TR-30
ESD-TR-88-31

Carnegie-Mellon University
Software Engineering Institute

An OOD Paradigm for Flight Simulators, 2nd Edition

Kenneth J. Lee
Michael S. Rissman
Richard D'Ippolito
Charles Plinta
Roger Van Scoy

December 1987, First Edition
September 1988, Second Edition

Technical Report

CMU/SEI-88-TR-30

ESD-TR-88-31

December 1987, First Edition

September 1988, Second Edition

An OOD Paradigm for Flight Simulators, 2nd Edition

Kenneth J. Lee

Michael S. Rissman

Richard D'Ippolito

Charles Plinta

Roger Van Scoy

Dissemination of Ada Software Engineering Technology

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

Preface

This is the second edition of the SEI Technical Report, *An OOD Paradigm for Flight Simulators*, which was first issued in December, 1987. We have issued this edition to report modifications we made to the paradigm while preparing for a tutorial given at the March, 1988 AdaJUG in Phoenix, AZ.

The paradigm is being used by SEI affiliates on full-scale devlopment programs. The SEI project team supports the use of the paradigm by consulting with the affiliates. The affiliates' efforts improve the paradigm by tailoring it to the nuances of particular programs.

This report does not describe all the ways the paradigm has been tailored to fit specific programs. The results of those efforts will be documented in subsequent versions of this report. Some of the more substantial changes are treated as Open Issues in Section 6 of this edition.

1. Introduction

1.1. Abstract/Background

This report presents a paradigm for object-oriented implementations of flight simulators. It is a result of work on the Ada Simulator Validation Program (ASVP) carried out by members of the technical staff at the Software Engineering Institute (SEI).

1.2. Motivation

Object-oriented design (OOD) predominates discussions about Ada-based software engineering. The identification of objects and the implementation of objects are two separate issues. This paradigm is a model for implementing systems of objects. The objects are described in a form of specification called an object diagram.[1] The paradigm is not about how to create the specification.

Although much has been written on object-oriented design, SEI project members could find no examples of object-oriented implementations relevant to flight simulators. Examples were required for two reasons. First, object-orientation was new to both of the contractors on the ASVP. A methodology which leads to a specification of objects is useful only if developers know how to implement what is specified. Second, managers were skeptical about the benefits of object-oriented design. Examples were needed to determine whether benefits outweigh costs.

The intent of our work was to produce examples of object-oriented systems. It was not our intent to determine whether object-oriented design was best for flight simulators.[2]

[1] See Chapter 4 and Figure 4-2 for an example of an object diagram.

[2] See Section 2.1 for some historical motivation.

1.3. Characteristics of the Application Domain

The paradigm was developed for a specific application domain, namely flight simulators and training devices. This section puts the paradigm in context by briefly describing the relevant features of the application domain.

The objective of a flight simulator is to reproduce on the ground the behavior of an aircraft in flight. Simulators are used to:

- train aircrew
- train maintainers of aircraft
- aid designers of aircraft

A training simulator consists of a mock-up of stations for the aircrew being trained. The mock-up contains the controls available to manipulate the aircraft and systems for cuing the operator to the aircraft's response to his actions. Cues include gauges, video, sound, and motion.

The training mission is set by an instructor at an Instructor Operator Station (IOS). Some of the factors set by the instructor are longitude, latitude, altitude, and atmospheric conditions. Instructors also affect the behavior of the simulator by introducing aircraft malfunctions.

The ASVP focused on software that models the behavior of major systems affecting an aircraft's flight: the airframe, the engines, the electrical system, the fuel system, the hydraulic system, and others.

Traditionally, this software is put under the control of an executive which periodically updates systems. Flight simulators are not event-driven. Interaction between systems in the real aircraft are continuous. Simulators model those interactions in discrete time.

Time constraints are normally tighter than memory constraints. Multiple processors are used to distribute processing and to link the software to hardware in the aircrew training station. Trends are such that multi-processor architectures are becoming more prevalent in the domain.

Flight simulators are long-lived and frequently modified. The two major modifications are changes to the aircraft itself which must be reflected in the simulator software and, secondly, changes in the training missions. Typical of the latter are the addition of new malfunctions.

Flight simulators are based on math models provided by the manufacturer of the aircraft components in the actual aircraft. The ultimate test of the simulator is the way it feels to aircrew experienced with the aircraft being simulated. The process of tuning the feel of the simulator is called aircrew tuning.

Flight simulators provide natural opportunities for reusing software. First, different aircraft have the same kinds of components, e.g., engines, fuel systems, electrical systems, etc.

Sometimes a particular instance of a kind of component, a Pratt and Whitney engine for example, is used on a variety of aircraft. Second, the three classes of simulators—training, maintenance, and engineering—model the same components to varying degrees of fidelity. Third, a simulator is made up of systems that can be viewed identically at some level of abstraction.

1.4. Reader's Guide

This report contains the work completed to date, presents the paradigm, and discusses the advantages of the paradigm. It is meant to stand on its own merits. The model we have developed solves a specific set of problems. We do not claim it to be the only model for solving these problems. The paradigm uses many of the characteristic software engineering concepts, but the report is not intended to be a report on software engineering theory.[3]

The next chapter discusses our approach to developing the paradigm and how we assessed the fit of our solution to the problem at hand.

Chapter 3
: introduces the conceptual elements of the paradigm and provides an overview of the software structure implied by the paradigm.

Chapter 4
: presents a detailed view of the elements of the paradigm. The elements are presented bottom-up using an Engine system as an example. Each section on a particular element ends with a discussion of the benefits of the implementation chosen for the paradigm. The final section of Chapter 4 summarizes the benefits of the paradigm.

Chapter 5
: discusses the role of a paradigm in the development process.

Chapter 6
: discusses issues which we have thought about during the development but have not had time to fully address.

Chapter 7
: is a very brief presentation of a simulator Electrical system.

Appendix A
: describes a modified form of the notation expounded by Grady Booch in his book on software engineering with Ada [1] and his book on reusable software components with Ada [2]. The notation is used in the diagrams in Chapter 3.

Appendix B
: contains an object manager template. The use of reusable code templates is discussed in Chapter 5.

Appendix C
: presents a version of the Engine system code complete through the package specifications. The intent is to demonstrate the software architecture defined by the object paradigm discussed in Chapter 4.

---

[3] If the audience perceives that this report would be useful within a tutorial on software engineering, we invite such a use of the report.

2. Approach

2.1. History

The project team began the search for a paradigm after reviewing an implementation of an electrical system done by one of the contractors on the ASVP. The implementation was more data-oriented than object-oriented. The implementation was a definite improvement over the original FORTRAN implementation. However, the team did not consider the implementation to be exemplary.

The project team decided to spend what it thought would be no more than a month or two developing an example of a pure object-oriented design of an electrical system. A circuit diagram was used to identify the objects and the relationships among the objects. The behavior of the objects, e.g., circuit breakers, relays, and batteries, and of circuits in general, was well understood.

Material available to us on object-oriented design did not adequately address connections among objects or updating systems of objects in discrete time.

The project team implemented an object-oriented electrical system which came close to satisfying the goals described below. At that time one of the contractors on the ASVP asked the project team to sketch out an object-oriented implementation of an engine. The team observed that the object-oriented implementation of an engine and of an electrical system were identical at some level of abstraction.

The project team decided to capture the similarities in a paradigm for object-oriented systems. The paradigm was to dictate how an object-oriented specification would be implemented in software and how the update of systems would be controlled. The drive to generalize uncovered flaws in our designs of both the engine system and the electrical system.

The project team did not develop the paradigm methodically. We were not interested in testing design methods. Our goal was to produce a paradigm for object-oriented systems. We did not want to limit our search space to architectures produced by known methods.

2.2. Design Goals

The project team began with two basic goals. One was to eliminate nested implementations of objects. The other was to simplify dependencies among objects.

2.2.1. Nested objects

Nested objects result from decompositional approaches that purport to help the designer discover which objects are needed to implement a system. For example, the designer begins with the notion of an engine as a black box. All interfaces to the engine appear at the surface of the black box. Now, suppose the vibration of an engine compressor needs to be metered. The designer decides to decompose the engine into other objects, one of which is a compressor. Access to the vibration level of the compressor passes through two levels: the engine level and the compressor level. Further, decomposition might lead to modeling each stage of the compressor as an object, thus adding a third layer to the nested object. Finally, black box implementations require knowledge of the entire black box, even when only one state or aspect of the black box is used.

Nested, hierarchical objects do have advantages. First, it should be possible to update a composite object, such as an engine, as if it were a black box. Second, it should be possible to reuse an object, such as an engine, as a separate entity.

2.2.2. Object dependencies

Figure 2-1 shows a dependency between objects A and B. In this example, B provides A with *something*.[4] Thus the state of A depends on the state of B.[5] There are several ways for handling this dependency. One common method is to have the implementation of object A *with* object B. When A is updated, A reads the relevant state of B. This method does not work if B and A are on separate processors. Even if A and B are on the same processor, it is never clear which object should define the dependent data type.

Another common solution is to have object B call object A and report its state. This solution introduces a new problem without solving the problem mentioned above. If the flow between B and A is continuous, then it is unnatural for object B to model discrete time by controlling the rate at which A is updated. Further, if B and A are part of a closed feedback loop, the update cycles indefinitely.

---

[4]The same diagrammatic notation is used throughout this report. The arrows represent dataflows. Thus *something* is needed by the object at the head of the arrow and is supplied by the object at the tail of the arrow. The arrow is labeled with the data near its tail.

[5]In Ada, an object which depends on another, separately compiled object, uses the *with* clause to gain visibility of the dependent object. The object is said to *with* the dependent object.

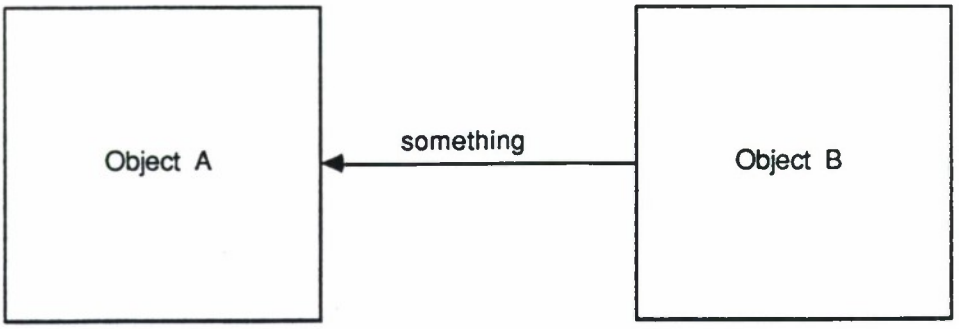


Figure 2-1: Object Dependency Example

2.3. Evolution of the Paradigm

Designers talk about the fit of a design to its context, the problem space. The criteria for assessing the fit of solutions to complex problems often can be determined only in response to a proposed solution and cannot be determined before solutions are generated. Such was the case for the paradigm.

Our team began with intuitive feelings about the standard goals of software engineering, such as modularity, ease of enhancement, and reuse. The paradigm passed through several iterations within the team. Each iteration left a legacy of criteria for assessing the fit of the solution for the paradigm.

For example, the model for object managers[6] and a means for connecting objects surfaced in the first version of the paradigm. The objects stood alone, and were not dependent on Ada types declared elsewhere. This enhanced the reusability of the object managers and facilitated independent development. The means for connecting objects had an intuitive analog in the real-world. Pipes and wires, connecting objects in the world, are as real as the objects themselves and would not be subsumed in software by the implementations of the objects.

Since the first edition, the notion of a connection has changed. Conduits, e.g., the **Engine Casing** in the Engine system (see Chapter 4), or wires in the Electrical system, can have states just like other objects and might need to retain information about previous states like other objects. Futhermore, these "connecting" objects could be in a malfunctioning operational state like other objects. Therefore, this kind of object is now considered equivalent to other objects. The abstraction which is now called a connection merely transfers data between any two objects. A connection does not have an analogy in the physical world; it merely implements an arrow on the object diagram, Figure 4-2.

[6]Object managers are introduced in Chapter 4.

Also, after issuing the first edition, we determined that the distinction we had made between systems and subsystems was not necessary. Thus, the notion of a subsystem has been eliminated.

The chapters which follow discuss the advantages of the paradigm. We did not set out to obtain these advantages. The advantages revealed themselves as the work progressed. An advantage which revealed itself in one iteration, was retained as a criterion for evaluating the fit of subsequent iterations.

3. Concepts Used by the Paradigm

This chapter provides a brief description of some of the concepts introduced with the paradigm and a high level overview of the software architecture defined within the paradigm. The concepts are further elaborated in Chapter 4.

The paradigm described in this report begins with the notion of an *executive*. An *executive* controls the update of a set of systems compiled together running on a single processor. The paradigm assumes that there will be more than one set of systems and that multiprocessing will be involved.

Communication between executives is handled by an abstraction called a *buffer*. A *buffer* is some means of sharing data among separately compiled software.[7] The paradigm makes no assumption about how the operating system transfers data or how executives on separate processors are invoked.

The fundamental units of the paradigm are *objects* and *connections*. *Objects* map to real-world entities. An *object* is implemented as a math model that maps the environmental effects on the object to the object's outputs, given the attributes of the object and its operational state. The implementation isolates individual effects. Also, an object is not aware of its connections to other objects.

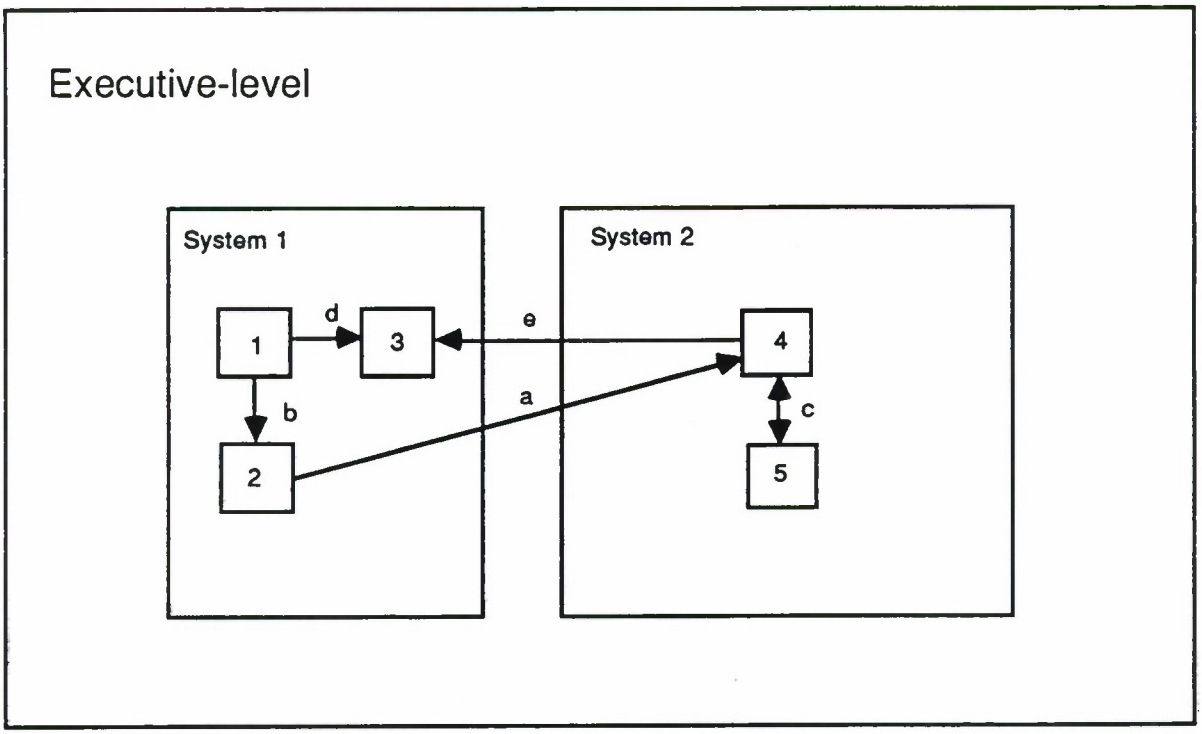
A *connection* is the mechanism for transferring state information between objects. Processing a connection involves reading the state of some objects on the connection and broadcasting to others.

At all levels, updates are accomplished by gating (or processing) the appropriate connections. The levels discussed in the paradigm are *system* and *executive*. A *system* is an aggregation of objects[8] and the connections among those objects. An *executive* is a set of systems and all connections that cross system boundaries, i.e., connections between objects in different sys-

[7] In our observations of flight simulators, a *buffer* is a record data structure used in the communication between processors.

[8] The aggregation is a matter of convenience. The objects are aggregated because they cooperate in performing common goals.

tems. Figure 3-1 shows views of an executive, two systems, and several objects and connections.



Executive is : System 1, System 2, and connections **a** and **e**
System 1 is : Objects1, 2, and 3, and connections **b** and **d**
System 2 is : Objects 4, 5, and connection **c**

Figure 3-1: Object Diagram Example

3.1. Overview of the Software Architecture

3.1.1. The Executive Level

Figure 3-2 shows the executive-level software architecture.[9] In this case, we assume an executive-level called Flight_Executive. The body of the Flight_Executive package contains a tabular schedule of systems to update. The names of the systems are declared in the package Flight_System_Names, the sole purpose of which is to enumerate the names.

Each system is represented by a package called **<system_name>_System.**[10] The specifi-

[9]See Appendix A for a description of the icons used in Figures 3-2, 3-3, and 3-4. The arrows on the diagrams represent (*withing*) dependencies. The shaded portions of each icon represent the package body, the white portions the package specification. Note that the dependencies originate within package bodies. This reduces the need for widespread recompilation in the event of a change.

[10]The use of "<...>" within subprogram names, type names, or text refers to a general case of the item. For example, **<system_name>_System**, is a general form representing all instances of the package name, e.g., **Engine_System, Electrical_System, Fuel_System**, etc. See Chapter 5 for a more detailed discussion and examples of the use of "<...>".

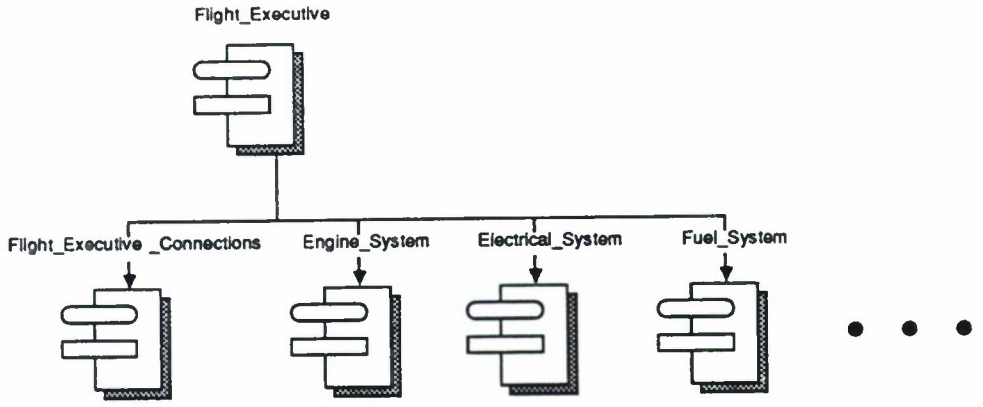


Figure 3-2: Executive Level Software Architecture

cation of a System package exports a single procedure which is called by Flight_Executive to update a system.

The connections belonging to the executive-level are managed by an **<executive_name>_Connections** package, in this case, Flight_Executive_Connections. The architecture from the perspective of the connection package is shown in Figure 3-3.

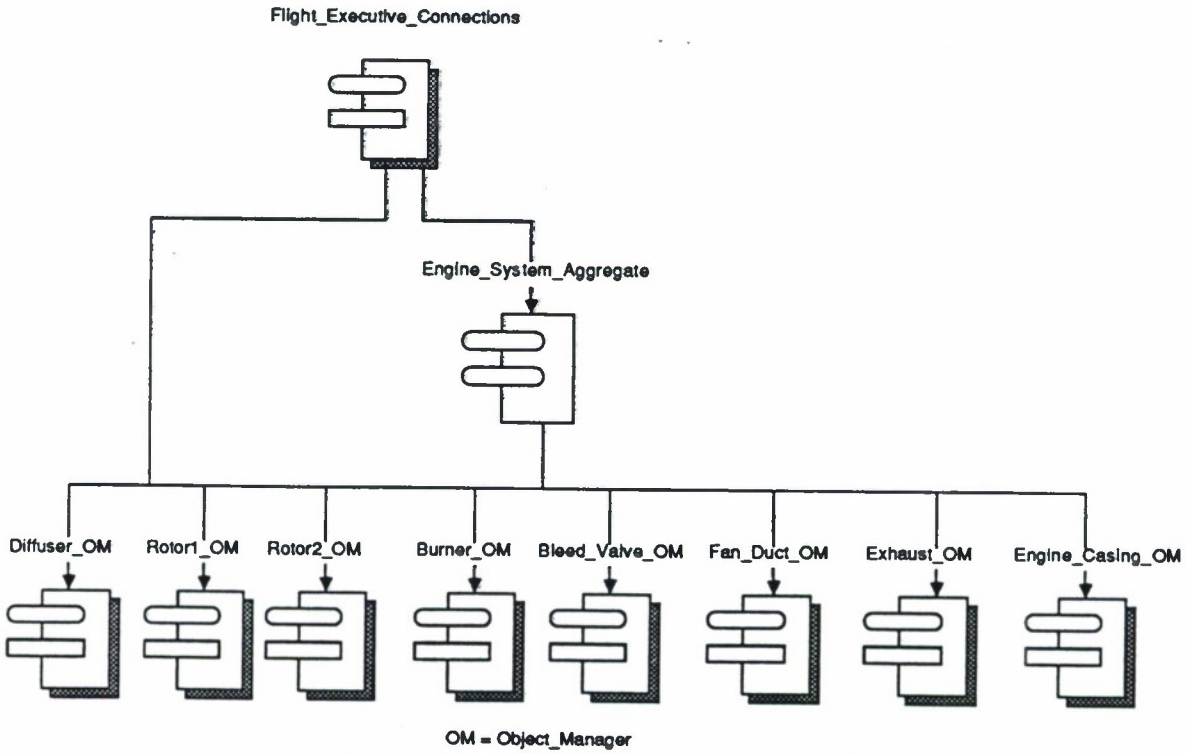


Figure 3-3: Connection Manager Software Architecture

The body of the connection package is a series of separate procedures, one for each system

under the control of the executive. Each separate procedure is responsible for gating all the executive-level connections to a system.

3.1.2. The System Level

Figure 3-4 shows the architecture from the perspective of a system, using the Engine system as an example. Objects in a system are created and named by the **<system_name>_System_Aggregate** package. Objects are managed by **<object_name>_Object_Manager** (OM) packages.

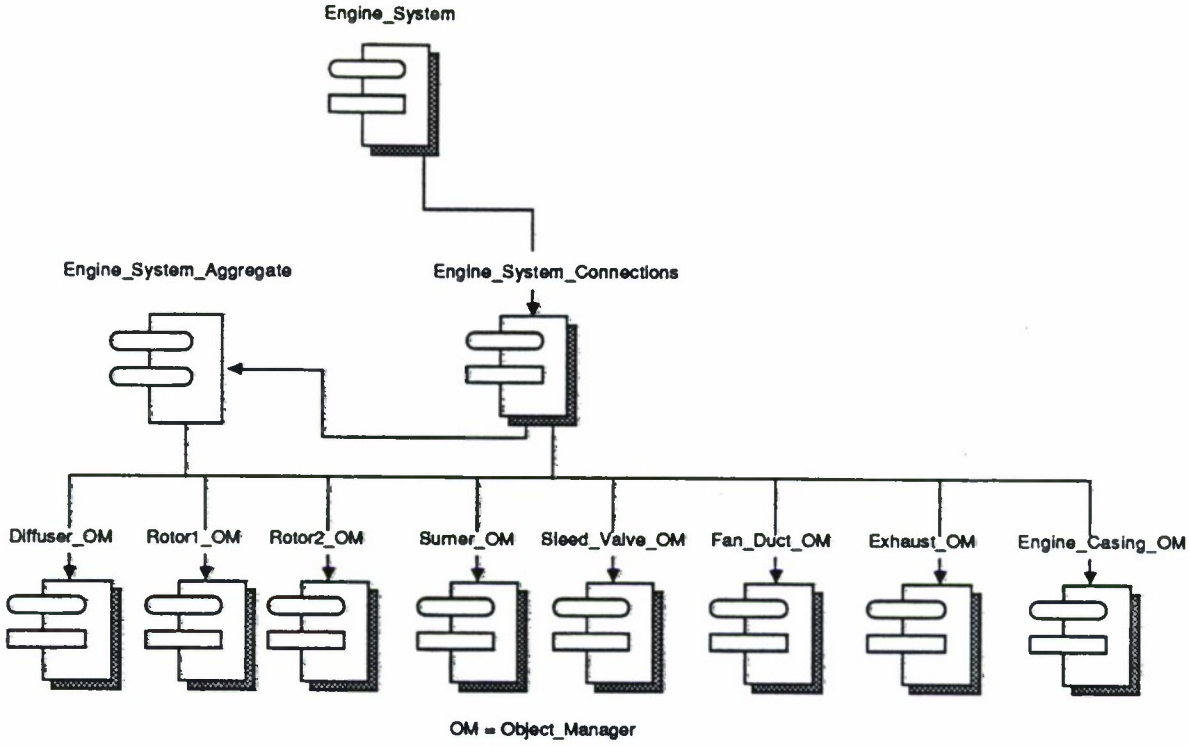


Figure 3-4: System Level Architecture

3.1.3. Overall Software Architecture

The overall software architecture is shown in Figure 3-5. The executive-level consists of the **Flight_Executive** package and the **Flight_Executive_Connections** package. The system-level consists of

- **<system_name>_System** package
- **<system_name>_System_Connections** package
- **<system_name>_System_Aggregate** package

The complete systel-level architecture of the Engine system is shown. The architecture of the other systems, e.g., the Fuel system and the Electrical system, would be similar.

Each connection package is "nested" within the corresponding system or executive packages.

Each connection within the connection packages is distinct, embodied within a separate procedure.

There is one object manager package per object.[11]

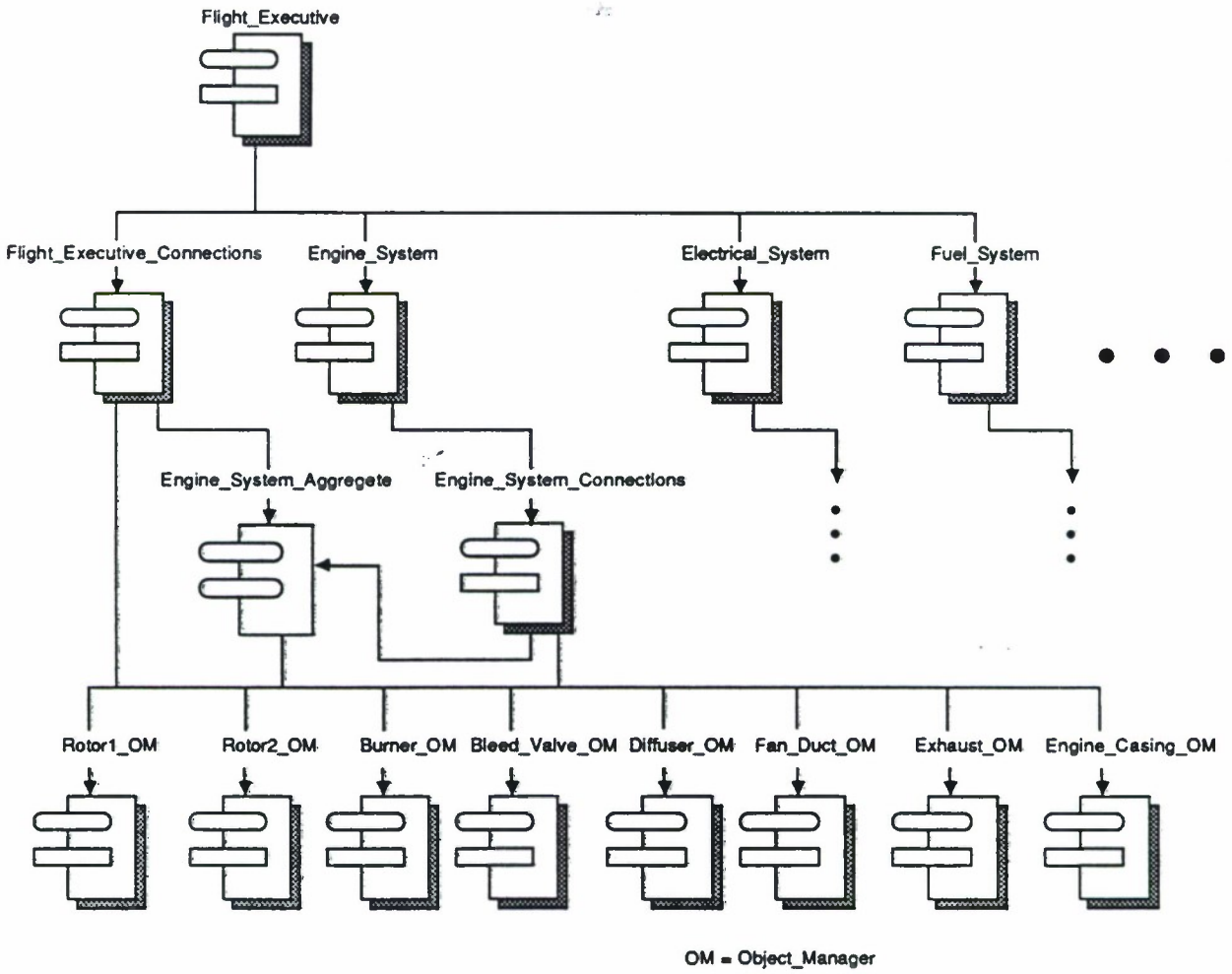


Figure 3-5: Overall Software Architecture

[11]Note the object managers have no dependency on other modules.

4. Paradigm Description

The example used to illustrate the paradigm is a turbofan engine. Engines, in flight training simulators, interact with a variety of other systems on the aircraft, including the Fuel system, the Oil system, the Starter system, the Electrical system, and the Hydraulic system. The engines also provide bleed air for Cabin Pressure and Air Conditioning systems.

Section 4.1 describes the engine components and the interaction of the engine with the other aircraft systems and the outside environment. The following section, Section 4.2, describes the Engine object diagram and, briefly, the meaning behind each icon on the diagram. The rest of the chapter introduces the paradigm by discussing the implementation of the Engine system.

4.1. Engine Description

Figure 4-1 shows a diagram of a turbofan engine with the relevant parts labeled. The **Engine Casing** encloses the other engine parts and provides the conduit through which the air flows as it interacts with the other parts. Air enters the **Diffuser** at some temperature, pressure, and flow rate. The **Fan Duct** directs part of the air flow out of the engine to provide some thrust to the Airframe system of the aircraft. The initial set of blades on the two **Rotors** adds energy to the air by compression. The **Burner**, or combustion chamber, adds more energy to the air by mixing fuel, from the Fuel system, with the air and igniting the mixture with a spark from the Ignition system. The second set of blades on the **Rotors** removes some energy from the air to turn the **Rotor** shafts and their initial set of blades. Finally, the **Exhaust** provides additional thrust on the Airframe system.

4.2. Engine Object Diagram

The Engine object diagram in Figure 4-2 is another representation of the Engine shown in Figure 4-1. All the functionality apparent from Figure 4-1 is evident in Figure 4-2. In addition, the Engine object diagram identifies the objects which comprise a generic turbofan engine and the engine's relationship with the outside environment. The correspondence between the real-world components of the engine and the object components in the object

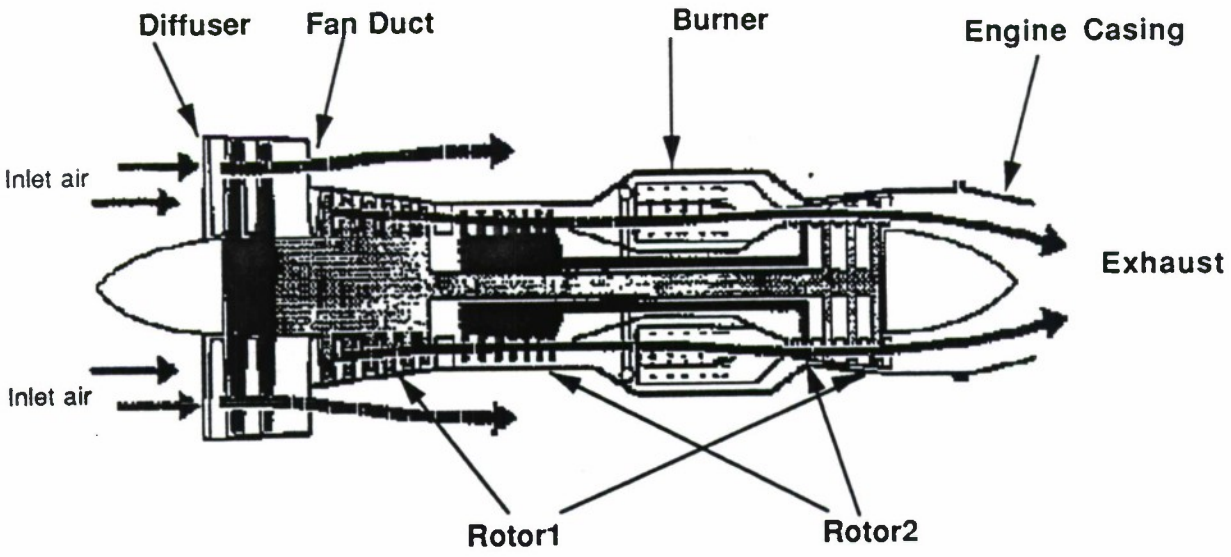


Figure 4-1: Turbofan Engine Description

diagram represents the first of two meanings for object-orientation in this solution.[12] The choice of objects may not be ideal but, for the purposes of the discussion in this report, this set of objects is acceptable. (The Engine object diagram, Figure 4-2, will be referred to throughout the rest of this chapter.)

There are four icons on the object diagram to represent four kinds of entities—objects, connections, systems, and executives.

The square boxes within the rectangle represent the engine objects. The objects are:

- **Diffuser**
- **Rotor1**
- **Fan Duct**
- **Rotor2**
- **Burner**
- **Bleed Valve**
- **Exhaust**
- **Engine Casing**

The function of the objects is to map their inputs to their outputs. Objects in the real-world know nothing of their environment, neither the objects which depend on them nor the objects upon which they depend. We model objects the same way.

[12]The second meaning for object orientation is described in Section 4.3.

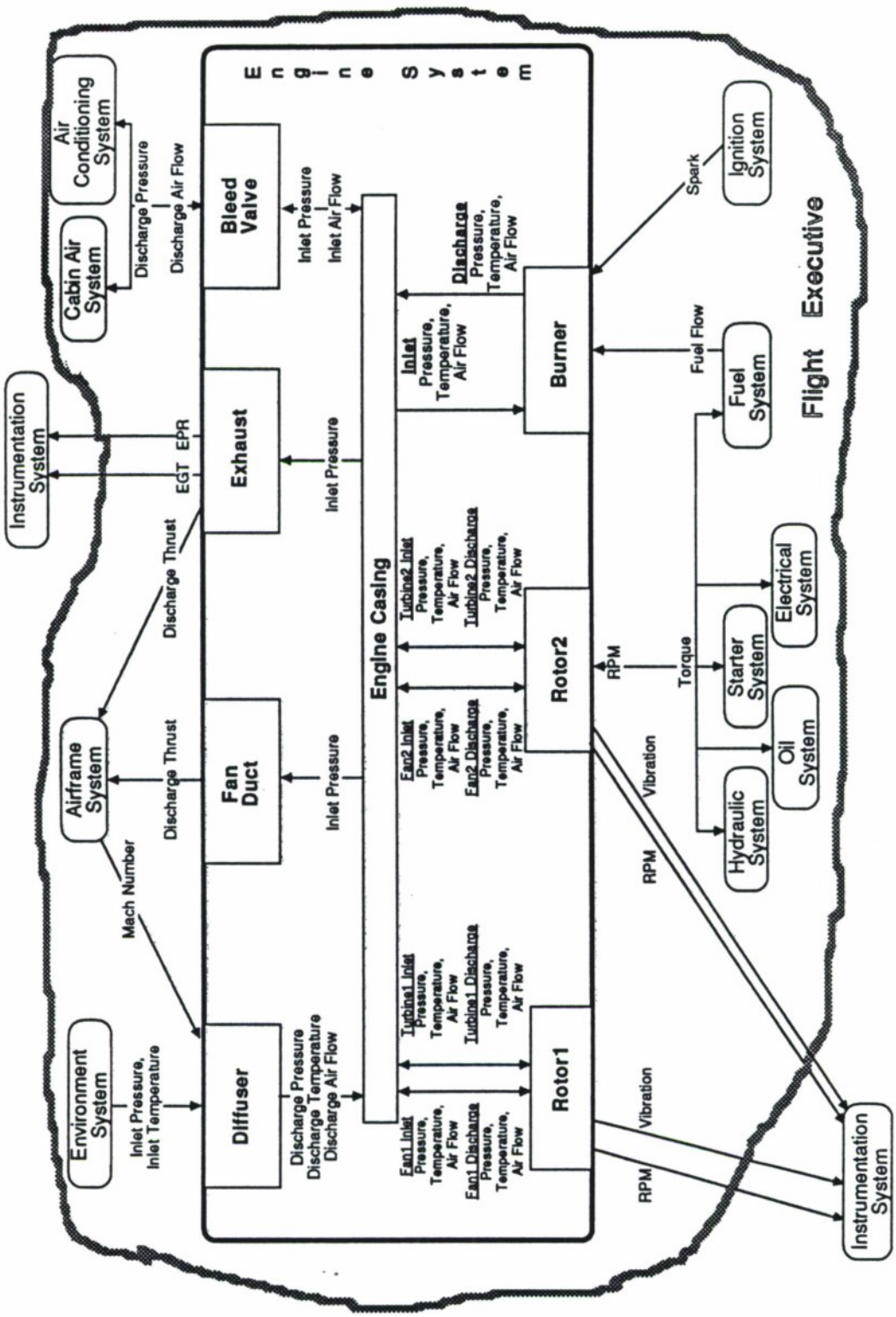


Figure 4-2: Turbofan Engine Object Diagram

The **Engine Casing** object is the object through which the air flows as the air passes through the engine. Each object has some dependency on the air flow, as it passes through the **Engine Casing**, denoted by the arrows between the **Engine Casing** and the other objects. Thus, the **Rotor1** *Fan1 Inlet* air pressure, temperature, and flow comes from the **Engine Casing**. Also, each object has outputs needed by the **Engine Casing**, e.g., the **Rotor1** *Fan1 Discharge* air pressure, temperature, and flow are available to the **Engine Casing**.

Each engine object in the Engine object diagram interacts with its external environment as defined by the diagram. No other dependencies on the outside world should be necessary except for those shown in the diagram. The diagram serves as a specification for the Engine system interfaces. Given such a diagram and the paradigm description that follows, the design of the Engine system is complete.

The arrows represent connections. A connection moves information between objects. An arrow points in the direction of data flow, e.g., a datum, called *mach number*, flows from the Airframe system to the **Diffuser** object, and other information flows from the **Exhaust** object to the Airframe system and the Instrumentation system.[14] A double-headed arrow represents two single-headed arrows, one pointing in each direction. The arrows are always labeled with the state information that is passed between the objects. The label nearest the tail of an arrow names the data flowing toward the head of the arrow.

On the Engine object diagram, the Engine system is the area within the large, round-cornered rectangle (roundtangle). The roundtangles external to the Engine system represent other systems in the aircraft, e.g., Electrical system and Fuel system, or in the aircraft's environment, e.g., the Environment system. A system is composed of its objects and the connections between the objects. These connections are called system-level connections. Thus, the Engine system is made up of the engine objects and the connections between them inside the roundtangle. An aircraft simulator for a multi-engine aircraft would have multiple engine systems. Each would be handled identically internally but would have different connections to the outside world.

A system provides two abstractions. First, a system logically groups a set of objects and their connections. Second, a system provides an update abstraction to update the objects as a unit in order to maintain system state consistency. The system performs the update by gating, i.e., processing, all of its system-level connections.

The final icon on the Engine object diagram is the executive, represented by the heavy, gray outline. An executive groups a community of systems and coordinates time for the community, i.e., provides an ordered update of all the systems. The connections between systems are executive-level connections. An ordered update of a system consists of two steps:

---

[14] In actuality, all connections are between objects. So, more correctly, the *mach number* flows from some object in the Airframe system to the **Diffuser** object in the Engine system. This point will be elaborated in later sections of this report.

- gating the executive-level connections and
- calling the system to perform its update.

The object diagram thus depicts natural, real-world entities, such as objects and systems, and entities that originate from the commitment to run the simulator on a computer, i.e., connections which move data and executives which control time and allocate resources, such as the CPU. Each of the abstractions—objects, connections, systems, and executives—will be discussed in more detail in the rest of this chapter.

4.3. Object Abstraction

This section describes an object abstraction assuming the objects are identified. The engine diagram in Figure 4-2 will serve as an example.

Objects correspond to real world entities. Objects generalize behavior, i.e., they know nothing about their environment and they are identical in each of the engines in a multi-engine system. They only differ in how they are connected to their environment. The objects, however, have no knowledge of these connections.[15]

A snapshot of the latest external effects is retained in the objects. The outputs (also called the *state* of the object), which are readable at any time, are always consistent with the latest snapshot. The function of the objects is to map from the inputs to the outputs.

4.3.1. Object Managers

Each object is represented by an object manager. There is a single object manager for all instances of the object.[16] Referring to the Engine object diagram, Figure 4-2, there will be an object manager for each of the objects in an engine:

- **Diffuser**
- **Rotor1**
- **Fan Duct**
- **Rotor2**
- **Burner**
- **Bleed Valve**
- **Exhaust**
- **Engine Casing**.

The object manager defines the *attributes* of the object. The attributes are invariant characteristics defined at elaboration, e.g., an ampere rating of a circuit breaker.

---

[15]Connections are described in Section 4.4.

[16]The term *manager* is used because all access to each object is administered through the interface defined by the object manager.

The object manager defines the *operational state* of the object. The operational state refers to those characteristics which may change with time, e.g., the frictional state of a rotor, malfunctions, or aging effects on various components.

The object manager allows the object's *environmental effects* to be placed on the object. The environmental effects are external object states which are required by the object to determine its state. The environmental effects are placed on an object by connecting procedures. The procedures defined for these operations are described in Section 4.3.3.

The object manager implements the math model for the object. The math model is implementation dependent. The math model maps the object's inputs to its outputs.

The object manager produces the *outputs* available from the object. The outputs are generated by the math model, using the environmental effects placed on the object and any additional constraints imposed by the attributes and the operational state of the object. The math model may be invoked when environmental effects are placed on the object or when outputs are read from the object. This is an implementation level decision left to the system designer; it is not defined by the paradigm.

The object manager defines an interface to the operations available on an object. The operations allow the placing of environmental effects, updating the operational state, and reading the outputs of the object.

The actual instances of the object are stored in object aggregates which are discussed in Section 4.5.1. An aggregate allows named access to the objects; no procedure call is required to retrieve the object.

Finally, the object manager is independent of the rest of the system. The only compilation dependencies are on global types.

4.3.2. Object Manager Structure

The representation of the object in an object manager is declared as a private type in the package specification.[17] Figure 4-3 is a partial package specification containing typical type definitions found in an object manager.[18] Use of a private type allows external access to the object, through the operations provided, while hiding the details of the object's implementation. In addition, the package specification must define all the types used to describe the object's attributes, the operational state, and the placeholders for environmental effects.

[17]For example, "type Burner is private" in Figure 4-3.

[18]Package **Standard_Engineering_Types**, *with*ed at the beginning of Package **Burner_Object_Manager** in Figure 4-3, contains global definitions for typical simulator types. The package is shown in Appendix Section C.2.

```
with Standard_Engineering_Types;

package Burner_Object_Manager is

   package Set renames Standard_Engineering_Types;

  type Burner is private ;
  -- an Burner is an abstraction of a Burner within an Engine.

  type Spark is (None, Low, High);
  -- burner needs only to know relative spark size

  type Fuel_Flow is (None, Flowing);
  -- the burner needs to know only if it has fuel available

  function New_Burner return Burner;

  procedure Give_Inlet_Air_To
        (A_Burner           : in Burner;
         Given_Inlet_Pressure   : in Set.Pressure;
         Given_Inlet_Temperature : in Set.Temperature;
         Given_Inlet_Air_Flow   : in Set.Air_Flow);

  procedure Get_Discharge_Air_From
               (A_Burner : in Burner;
                Returning_Discharge_Pressure   : out Set.Pressure;
                Returning_Discharge_Temperature : out Set.Temperature;
                Returning_Discharge_Air_Flow   : out Set.Air_Flow);

  procedure Give_Fuel_Flow_To
               (A_Burner     : in Burner;
                Given_Fuel_Flow : in Fuel_Flow);

  procedure Give_Spark_To (A_Burner   : in Burner;
                           Given_Spark : in Spark);

pragma Inline (Give_Inlet_Air_To,
               Get_Discharge_Air_From,
               Give_Fuel_Flow_To,
               Give_Spark_To)

private
  type Burner_Representation;
  -- incomplete type, defined in package body

  type Burner is access Burner_Representation;
  -- pointer to an Burner representation

end Burner_Object_Manager;
```

Figure 4-3: Burner_Object_Manager Package Specification

[19]For the **Burner** Object Manager in Figure 4-3, type definitions for *Spark* and *Fuel_Flow* are provided. In the private part of the package specification, the object's private type is

[19]The attributes and operational state variables must be visible to the system aggregate package which instantiates the object and to the system-level and executive-level connections packages which use the object's types and operations. See Sections 4.4, 4.5, 4.5.1, and 4.6 for descriptions of connections, systems, aggregates, and executives, respectively.

declared as an access pointer to a data type which will be the actual representation of the object. The data type is an incomplete type, the details of which are delayed until the package body.[20]

The object's data representation, defined in the package body, must allow for storage of environmental effects and reading of the object's outputs. A typical implementation is a record with components for each of the object's attributes, operational state variables, and placeholders for the environmental effects. Each attribute component must have a default value and each operational state variable should have an initial state value. Figure 4-4 contains an incomplete package body for the **Burner** object manager. The *Burner_Representation* is a record with fields for environmental effects, e.g., inlet air pressure, temperature, and flow, fuel, and spark values. The record also has fields for output values, e.g., discharge pressure, temperature, and flow.

```
package body Burner_Object_Manager is

   type Burner_Representation is
      record
         Inlet_Air_Pressure    : Set.Pressure    := 0.0;
         Inlet_Temperature     : Set.Temperature := 300;
         Inlet_Air_Flow        : Set.Air_Flow    := 0.0;
         The_Spark             : Spark           := High;
         The_Fuel              : Fuel_Flow       := Flowing;
         Discharge_Air_Pressure : Set.Pressure   := 0.0;
         Discharge_Temperature : Set.Temperature := 300;
         Discharge_Air_Flow    : Set.Air_Flow    := 0.0;
      end record;

--
-- Subprogram bodies go here
--
end Burner_Object_Manager;
```

Figure 4-4: Burner_Object_Manager Package Body

4.3.3. Object Manager Operations

There are three types of operations within each object manager. There is also a standard naming convention for these operations. One side effect of the naming convention is that all object managers begin to look very similar. The similarity can be exploited to create an object manager template, see Chapter 5, which can be used to generate new object managers.

The first type of operation is used to create new instances of the object. This operation is a function, named **New_<object>**[21], which returns an instance of the private type, **<object>**.

---

[20]See Appendix Section C.4 for the complete Package Specification for the **Burner** object. Appendix C provides an implementation of the Engine system through the Ada specifications.

[21]The use of "<...>" within subprogram names, type names, or text refers to a general case of the item. For example, **New_<object>**, is the general form representing all instances of the *New* function, e.g., **New_Burner**, **New_Rotor1**, **New_Exhaust**, etc. See Chapter 5 for a more detailed discussion and examples of the use of "<...>".

For example, in Figure 4-3, the function provided by the **Burner** object manager is called **New_Burner**; it returns an instance of the private type, **Burner**. This private type is a pointer to a new instance of the data type representing the object.[22] In addition, values for attributes or operational state variables, which need their default values changed or their initial values defined, may be set by the **New_<object>** function. Typically, this function is called at elaboration, i.e., during system initialization. The return value, a pointer which is the "ID" of the new object, is stored and used to access the object in later operations. See Section 4.5.1 for more discussion on this point.

The second type of operation is used to write external effects, i.e., environmental effects and operational state changes, on an object. The naming convention for this operation is **Give_<external_effects>_To**. The operation takes the object private type and either external environment values or new operational state values as arguments. In Figure 4-3, the procedure **Give_Inlet_Air_To** is an example of this type of operation.

The characteristics of the **Give_<external_effects>_To** procedure are as follows:

- report external environmental effects to the object. The stored values of the environmental effects will be used the next time the object's outputs are calculated. These updates are typically under the control of a cyclic executive and are placed on the object one or more times each cycle.
- report a change in the operational state to the object. The stored values of the operational state variables will be used the next time the object's outputs are calculated. These changes are typically asynchronous events triggered by the instructor at the IOS.
- the environmental effects and operational state variables are "saved" with the object in the private data structure.
- the environmental values stored with the object are consistent with the external effects at all times.

Ideally, the math model isolates the individual effects of the environmental effects. Calculation of the object's outputs can be postponed until the object's internal state is read.

The interfaces defined by the **Give_<external_effects>_To** operations can be read directly off the object diagram, Figure 4-2. There will be one procedure per dataflow arrow. For example, in Figure 4-3, procedure **Give_Inlet_Air_To**, for the **Burner** object manager, takes the pressure, temperature, and air flow as arguments.

The third type of operation is used to read an object's outputs. The outputs are calculated by the math model using the environmental effects placed on the object and any additional

---

[22]There are other options for managing storage allocation for the objects. One is to use the allocator directly, within the system aggregate package, rather than performing the function call to **New_<object>**. But then the type **<object>_Representation** would have to be visible. A second method would be to build an alternate allocator using statically defined **<object>_Representations**. Then each time an **<object>** had to be allocated, one of the statically defined instances would be assigned. This approach has merit if garbage collection is an issue. We are continuing to look into these and other approaches.

constraints imposed by the attributes and the operational state of the object. The naming convention for this operation is **Get_<object_output>_From**. The operation takes the object private type as an argument and returns the object's outputs. In Figure 4-3, the procedure **Get_Discharge_Air_From** is an example of this type of operation.

The characteristics of the **Get_<object_output>_From** operation are as follows:

- the response reflects the current state of the object. The state is dependent on the environmental effects previously placed on the object, the object's attributes, and the object's operational state. The outputs are read from the private data structure or calculated from the values stored in the data structure.
- the output state of the object is consistent with the external environmental effects at all times
- each operation is specific to the object and the output of the object that it reports. This operation is the only way to access the object's output.

The interfaces defined by the **Get_<object_output>_From** operations can be read directly off the object diagram, Figure 4-2. There should be one procedure per dataflow arrow. For example, in Figure 4-3, procedure **Get_Discharge_Air_From**, for the **Burner** object manager, returns the pressure, temperature, and air flow.

The output state of an object, determined from its environmental effects, attributes, and operational state, may be calculated either when new external information is written to the object (and then the output state should be stored with the object), by the **Give_<external_effects>_To** procedure, or when outputs are read from the object, by the **Get_<object_output>_From** operation. In the first case, each time an external effect is deposited, a new output state should be calculated and stored so that the correct output state can be returned on subsequent read operations. Since each external effect is independent of all others, the object's output state will be consistent at all times. In the second case, an object's output state is not stored, but calculated each time the outputs are read. The decision as to which implementation to use is up to the implementor of the system. That level of detail is not specified in the paradigm.

4.3.4. Advantages of the Object Abstraction

The object abstraction developed here is the second of two meanings of object orientation.[23] The object abstraction includes:

- the packaging strategy used, i.e., private types and local data stored with the object
- the object operations which are intentionally designed without side effects
- the objects which are stand-alone with no dependencies on other entities in the solution

Thus, there is a natural progression from real-world entities to design objects and from design objects to a consistent software representation.

---

[23]The first was the correspondance between real-world entities and design objects on page 17.

The implementation of objects follows the standard model for object-oriented abstraction. The object managers embody the state of objects, and changes in the objects' environment are placed on the objects procedurally. The major difference is the removal of connections from the objects (connections are described in Section 4.4). This decision supports separate development of objects since there is no dependency on any modules other than global types. In addition, spaghetti compilation dependencies are prevented. Finally, reuse is supported, since data-type differences between objects are not an issue.[24]

Another advantage of the object managers is to focus the addition of details in one place. For example, if there is loss of efficiency in the movement of air through the **Burner**, the loss can be modeled in the object manager for the **Burner**. Also, malfunctions in components can be simulated in the objects. The introduction, handling, and reporting of a malfunction should be introduced at the object manager level.

4.4. Connection Abstraction

In the real-world, laws of nature or physics govern the transfer of state information between objects. For example, heat provided by the **Burner** is transferred to the air flowing through it. Futhermore, the laws of nature function continuously on a single "processor" without regard for units of measurement or other information.

In a computer system, state information must be transferred explicitly among objects that are updated in discrete time on multiple processors and must be transferred with some type of units.

This section describes connections, the mechanism for transferring state information between objects. Connections do not correspond to real-world entities such as wires or pipes. Connections simply model the proximity of one object to another in the real-world.

4.4.1. Overview of Connections

The connections in Figure 4-2 are represented by arrows. An arrow points in the direction of data flow. A double-headed arrow represents dataflow in both directions.

Connections also provide a means to transfer information between physical objects and software objects. Buffers can exist between physical objects and the software system. The buffers may be, for example, a linkage buffer between the software and the simulator hardware, an Instructor Operator Station (IOS) buffer between the software and the IOS station, or buffers between processors in a multi-processor configuration. In all these cases, the connection handles the transfer of environmental effects or operational state information from the buffer (the representation of the physical object) to the software objects and the transfer of object state from the software objects to the buffer. For example, software lights in the electrical system can be turned on and off as a result of external environmental effects

[24] One of the roles of connections is to convert types when necessary, see Section 4.4.

or operational state changes. These effects must be transferred to the simulator cockpit and affect a change in the physical lights. Lights can also be turned on and off in the simulator cockpit by the students. These effects must be transferred to the software and change the operational state of the software lights. The linkage buffer between the cockpit and the software is used and connections handle the information flow.

Finally, the updating of a system is accomplished by moving information along connections , i.e., gating the executive-level and system-level connections in order.

4.4.2. Procedural Abstraction

The connections between objects are captured procedurally, using the object operations. All connections between objects within systems and between systems are modeled this way. These operations, defined with the object, allow for writing information to the object and reading information from the object. See Section 4.3.3 for more discussion on the object operations.

Thus, the connecting procedures exist outside the object managers, but have visibility into the object managers.

The connecting procedures need to perform three steps:

- obtain the needed information directly from an object
- convert the information if necessary
- put the information directly onto another object

Each step is discussed in more detail in the following sections.

4.4.2.1. Get Needed Information

The initial step is to obtain the external information which must be placed on an object. The provider of the information is defined within an object diagram at the tail of each arrow, as in the Engine diagram, Figure 4-2. The provider will be either an object in an external system, e.g., the Fuel system or Ignition system, or another object within the Engine system.

If the provider is from an external system, the procedure modeling the connection must have access into the objects of each system. Thus the procedure needs to exist at the next higher level of abstraction, i.e., within the enclosing executive. These connections are called executive-level connections. Within the executive connection procedure, local variables may exist to allow for temporary storage of the information, as in Figure 4-5. The current value of *spark*, from the **Ignition** object manager, is obtained with a call to **Get_Spark_From** and stored in the local variable *Some_Spark*. Thus, although the paradigm does not advocate careless data-typing, it recognizes that perfect type matches between objects will not always be possible.

If the provider is from another object within the Engine system, then the enclosing scope of the objects, i.e., the Engine system itself, handles the connection. These connections are called system-level connections. Figure 4-6 shows the connection between the **Diffuser** and

```
with Standard_Engineering_Types;
with Engine_System_Aggregate;
with Ignition_System_Aggregate;

with Flight_System_Names;

with Burner_Object_Manager;
with Ignition_Object_Manager;

separate (Flight_Executive_Connection_Manager)

procedure Process_External_Connections_To_Engine_System is

  Integrated_Drive_Energy : Generator_Object_Manager.Energy;

  Some_Spark : Ignition_Object_Manager.Spark;
  The_Burner_Spark : Burner_Object_Manager.Spark;

  function Spark_Conversion (In_Spark : in Ignition_Object_Manager.Spark)
                return Burner_Object_Manager.Spark is
  begin
    case In_Spark is
      when 0 .. 2 =>
        RETURN Burner_Object_Manager.None;
      when 3 .. 9 =>
        RETURN Burner_Object_Manager.Low;
      when 10 .. 20 =>
        RETURN Burner_Object_Manager.High;
    end case;
  end Spark_Conversion;

begin  -- Process_External_Connections_To_Engine_System

  for An_Engine in Flight_Systems_Names.Aircraft_Engines loop

       Some_Spark := Ignition_Object_Manager.Get_Spark_From
           (A_Ignition => Ignition_System_Aggregate.Ignitions
                                  (Engines_To_Ignition_Map (An_Engine)));

       The_Burner_Spark := Spark_Conversion (Some_Spark);

       Burner_Object_Manager.Give_Spark_To
           (A_Burner   => Engine_System_Aggregate.Engines
                                  (An_Engine).The_Burner,
            Given_Spark => The_Burner_Spark);
   end loop;
end Process_External_Connections_To_Engine_System;
```

Figure 4-5: Executive-Level Connection -- Spark Conversion Routine

the **Engine Casing**. The discharge air pressure, temperature, and flow are obtained from the **Diffuser** with the call to **Get_Discharge_Air_From**.

4.4.2.2. Convert Information

The connecting procedures encapsulate type conversions. Each object manager maintains the state of the object in the units which make sense to that object. The connecting procedures handle the type conversions which are necessary between the object managers.

```
with Flight_System_Names;
with Engine_System_Aggregate;

with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;

package body Engine_System is

  procedure Update_Engine_System is

    Diffuser_Discharge_Pressure   : Set.Pressure;
    Diffuser_Discharge_Temperature : Set.Temperature;
    Diffuser_Discharge_Air_Flow   : Set.Air_Flow;

  begin
       for An_Engine in Flight_System_Names.Aircraft_Engines loop

    --
    -- Model the connection characterized by the dependence of the
    -- Engine Casing on the Diffuser for Pneumatic_Energy.
    --
            Diffuser_Object_Manager.Get_Discharge_Air_From
                (A_Diffuser =>
                     Engine_System_Aggregate.Engines
                          (Given_Engine_Name).The_Diffuser,
                 Returning_Discharge_Pressure =>
                     Diffuser_Discharge_Pressure,
                Returning_Discharge_Temperature =>
                     Diffuser_Discharge_Temperature,
                Returning_Discharge_Air_Flow =>
                     Diffuser_Discharge_Air_Flow);

            Engine_Casing_Object_Manager.Give_Air_Flow_To
                (A_Engine_Casing =>
                     Engine_System_Aggregate.Engines
                          (Given_Engine_Name).The_Engine_Casing,
                     Given_Air_Flow  => Diffuser_Discharge_Air_Flow);
            --
            -- .
            -- .
            -- .
            --
       end loop ;
  end Update_Engine_System;

end Engine_System;
```

Figure 4-6: System-Level Connection

In Figure 4-5,[25] the intermediate value, *Some_Spark*, obtained during the get information step above, is converted to the local variable *The_Burner_Spark* by the function **Spark_Conversion**. *The_Burner_Spark* is defined in terms of the spark type in the **Burner** object manager.

[25]The notation used in Figure 4-5, **Engine_System_Aggregate.Engines (An_Engine).The_Burner**, is part of the Engine Aggregate nomenclature discussed in Section 4.5.1.

There are two reasons for managing type conversions within the connection procedure. First, the object managers are then free from inter-object type dependencies. The object managers become stand-alone, with no dependencies other than on global data types. Thus, the object managers become reusable units. Separate development of the object managers is also supported. The second reason is that each object manager has a different need. There is no reason to expect that the **Burner** object manager would have a need to know how the **Ignition** object manager maintains the *spark* state. For example, the *spark* from the **Ignition** object manager may be in volts while the **Burner** maintains the value as an enumerated type (see Figure 4-5).

4.4.2.3. Put Converted Information

The final step is to place the external environmental information on the object being updated. The information must be in the proper type to match the dependent object. Once again, a picture, like that in Figure 4-2, defines the destination for the environmental information. The procedures **Give_Spark_To**, in Figure 4-5, and **Give_Air_Flow_To**, in Figure 4-6, are examples of put information operations.

4.4.3. Advantages of Connections

The implementation of connections in connecting procedures, as described in this chapter, provides a consistent and natural interface to the objects.

The connections insulate the objects from compilation dependencies. Objects and systems become stand-alone. Each can be developed independently. Connecting procedures provide a firewall: changes in implementation to objects on one side of a connection do not affect the implementation of objects on the other side.

Connections facilitate independent development and reuse. In particular, connecting procedures provide a systematic way to handle typing mismatches. The type conversions between objects are easily managed since the connecting procedures have visibility into the objects.

Connecting procedures provide a consistent means of updating systems and objects. Thus, connecting procedures provide a means for specifying control flow. No extraneous concepts or operations are required.

Finally, the connecting procedures provide a locus of control since all connections at an abstraction level are handled in one place.

4.5. System Abstraction

To this point we have defined objects and the connections between them. This section discusses a method for grouping the objects and connections together into a logical scope.

A system is an aggregation of objects (and the connections between the objects) with a common goal. For example, the objects making up the Engine system provide thrust; the objects of an Electrical system provide power. The system (objects and connections) is updated as a single entity.

Thus, a system presents two abstractions. The first is the aggregation of objects accessible by name outside the system, as discussed below. The second is the set of connections between the objects of the system; these system level connections are not visible to the executive level. The set of connections allows for an ordered update of the system as a unit.

A system update requires nothing more than gating the system level connections as described in Section 4.4. Objects outside the system are not accessed during the system update. The update is initiated by a procedure call from the executive to the system.

4.5.1. System Aggregates

A real-world system usually consists of collections of objects. An aggregate creates and names a collection of objects. An aggregate is a data structure containing the name of each object. Objects are accessed by name. A procedure call is not required to obtain a "pointer" to an object being accessed.

4.5.1.1. Building an Aggregate

As was described in Section 4.1, an engine is a collection of objects, including the diffuser, rotors, a burner, and so forth. Each object is managed by its own object manager. An engine record can be constructed as a grouping of these objects (see the **Engine_Representation** in Figure 4-7).

```
with Burner_Object_Manager;
with Bleed_Valve_Object_Manager;
with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;
with Exhaust_Object_Manager;
with Fan_Duct_Object_Manager;
with Rotor1_Object_Manager;
with Rotor2_Object_Manager;

package Engine_System_Aggregate is

  type Engine_Representation is
    record
      -- Defines an engine representation as consisting of:
      The_Diffuser     : Diffuser_Object_Manager.Diffuser;
      The_Rotor1       : Rotor1_Object_Manager.Rotor1;
      The_Fan_Duct     : Fan_Duct_Object_Manager.Fan_Duct;
      The_Rotor2       : Rotor2_Object_Manager.Rotor2;
      The_Bleed_Valve  : Bleed_Valve_Object_Manager.Bleed_Valve;
      The_Burner       : Burner_Object_Manager.Burner;
      The_Exhaust      : Exhaust_Object_Manager.Exhaust;
      The_Engine_Casing : Engine_Casing_Object_Manager.Engine_Casing;
    end record;

end Engine_System_Aggregate;
```

Figure 4-7: Engine Representation Example

For an aircraft as a whole there may be several engines. Using a constant array, an aggregate of the engines can be created which stores references to **Engine_Representations**, one

for each engine on the aircraft (see Figure 4-8). The constant array, **Engines**, is created at elaboration time. Each object is instantiated by a call to the function **New_<object>**, described in Section 4.3.3, with all initial conditions set by default. The pointer to the private type returned by the function is stored with the name of the object. Thus, reference to the object can be done by name. The aggregate data structure is visible so no procedure call is required to retrieve an object. The array is indexed by the enumerated engine names **Engine_1..Engine_4**. The engine names are defined in a global type package that defines all the system names.

The constant array, **Engines**, is defined in a package specification to allow access to the Engine system by an external system which *with*s the package and the appropriate object managers. The aggregate and object managers are used by the connecting procedures, discussed in Section 4.4.3, to reference the necessary objects. All references to objects are done through the aggregates. An object in an engine is referenced as:

```
Engines(Engine_Name).The_<object>
```

For example, the **Diffuser** of Engine 1 is referenced as:

```
Engines(Engine_1).The_Diffuser
```

and the **Rotor1** of Engine 3 is referenced as:

```
Engines(Engine_3).The_Rotor1
```

The code fragment, in Figure 4-6, shows how to reference an Engine object using the Engine aggregate. The *Discharge Air* is read from the **Diffuser** object using the reference, **Engine_System_Aggregate.Engines (Given_Engine_Name).The_Diffuser** and written to the **Rotor1** object using the reference, **Engine_System_Aggregate.Engines (Given_Engine_Name).The_Rotor1**.

4.5.2. Updating

The existence of systems allows the processing of the enclosed objects to be done as a unit. The process of updating a system occurs in two steps (Figure 4-10):

- the executive processes the executive level connections, see Section 4.6.1)
- the system processes the system level connections

The operations are done atomically for each system. This means that when it is time to update a system, all work necessary to complete both steps of the update is finished before the process is begun on another system.

Processing the executive level connections involves gating the connecting procedures, as described in Section 4.4. The external effects, i.e., effects from objects external to the system being updated, are placed on the system objects by the connecting procedures at the executive level.

Once all the external effects have been placed on the system objects, then the system level update is initiated by a single procedure call, **Update_<system_name>_System**, to the system.

```
with Flight_System_Names;
with Burner_Object_Manager;
with Bleed_Valve_Object_Manager;
with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;
with Exhaust_Object_Manager;
with Fan_Duct_Object_Manager;
with Rotor1_Object_Manager;
with Rotor2_Object_Manager;

package Engine_System_Aggregate is

  Engines : constant array (Flight_System_Names.Aircraft_Engines) of
              Engine_Representation :=
    (Flight_System_Names.Engine_1 =>
      (The_Diffuser   => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1     => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct   => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2     => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner     => Burner_Object_Manager.New_Burner,
       The_Exhaust    => Exhaust_Object_Manager.New_Exhaust,
       The_Engine_Casing =>
                 Engine_Casing_Object_Manager.New_Engine_Casing),

     Flight_System_Names.Engine_2 =>
      (The_Diffuser   => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1     => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct   => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2     => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner     => Burner_Object_Manager.New_Burner,
       The_Exhaust    => Exhaust_Object_Manager.New_Exhaust,
       The_Engine_Casing =>
                 Engine_Casing_Object_Manager.New_Engine_Casing),

     Flight_System_Names.Engine_3 =>
      (The_Diffuser   => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1     => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct   => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2     => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner     => Burner_Object_Manager.New_Burner,
       The_Exhaust    => Exhaust_Object_Manager.New_Exhaust,
       The_Engine_Casing =>
                 Engine_Casing_Object_Manager.New_Engine_Casing),

     Flight_System_Names.Engine_4 =>
      (The_Diffuser   => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1     => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct   => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2     => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner     => Burner_Object_Manager.New_Burner,
       The_Exhaust    => Exhaust_Object_Manager.New_Exhaust,
       The_Engine_Casing =>
                 Engine_Casing_Object_Manager.New_Engine_Casing));
end Engine_System_Aggregate;
```

Figure 4-8: Engine Aggregate Example

For the Engine system example, the external connections which need to be processed are

those from systems outside the Engine system, e.g., the Fuel system. These connections are handled at the executive level. Then the Engine system is updated, via the procedure **Update_Engine_System** (Figure 4-10). Performing the system level update involves processing the connections at the system level. The connection representing the dependency of the **Rotor1** on the **Engine Casing** for air flow, temperature, and pressure is shown in Figure 4-6. The update procedure is dependent on the Engine Aggregate and the object managers.

4.5.3. Advantages of Systems

The implementation of systems, as described in this chapter, encapsulates objects and connections within a logical scope. A system needs to access only its aggregated objects, the global types used by the objects, and the system level connections.

This separation of concerns allows for several things:

- reduction of the impact of compilation dependencies. Systems become standalone. Connecting procedures provide a firewall; changes in implementation to objects in a system on one side of a connection do not affect the implementation of objects in another system on the other side.
- separate development of components and reuse. Systems are self-contained. The only dependencies are on global types and object managers.
- potentially easy disbursement within a multi-processor environment (more on this in Section 6.1).

4.6. Executives

An executive is a community of systems. For example, the Flight Executive contains the Engine system, the Electrical system, the Fuel system, etc. The executive controls the updating of all the systems within its scope. The executive handles all connections between its systems, e.g., those between the Engine system and the Fuel system. In a multi-processing environment, in this model, there would be one executive level per processor. The executive would have access to buffers for communication between the processors. However, the synchronization among the processors would happen outside the executive.[26]

4.6.1. Implementation of an Executive

All the systems within the executive's scope are known to the executive, as are all the objects in those systems. The executive has an activity table,[27] indexed by system names, which defines a processing order for those systems. An implementation for use within a cyclic executive is shown in Figure 4-9. The constant array, **Its_Time_To_Do**, defines the frame

[26]For this domain, in order to meet the required deterministic real-time schedule, Ada tasking is not a viable solution. In our view, the executive functions like an abstraction of a CPU. The scheduler, that is shown in Figure 4-9, replaces the Ada tasking model at run-time.

[27]The nature of the activity table is not a concern of the paradigm. More elegant and powerful implementations are possible.

```
with Global_Types;
with Flight_System_Names;

package body Flight_Executive is

   type Active_In_Frame is
     array (Flight_System_Names.Name_Of_A_Flight_System) of Boolean;

   Its_Time_To_Do : constant array (Global_Types.Execution_Sequence) of
                    Active_In_Frame :=
     (Global_Types.Frame_1_Modules_Are_Executed =>
       (Flight_System_Names.Engine => (True), others => (False)),

      Global_Types.Frame_2_Modules_Are_Executed =>
       (Flight_System_Names.Electrical => (True), others => (False)),

      Global_Types.Frame_3_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_4_Modules_Are_Executed => (others => (False)),

      Global_Types.Frame_5_Modules_Are_Executed =>
       (Flight_System_Names.Engine => (True), others => (False)),

      Global_Types.Frame_6_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_7_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_8_Modules_Are_Executed => (others => (False)));

end Flight_Executive;
```

Figure 4-9: Executive Activity Table Example

in which each system, e.g., the Engine system and the Electrical system, gets processed. The processing is actually initiated by the procedure shown in Figure 4-10.

The updating of a system involves writing the external effects on the system and then telling the system to update itself. These operations for the systems are done atomically. For example, in Figure 4-10, when it is time to update the Engine system, a call is made to **Flight_Executive_Connections.Process_Engine_Connections_To**. This procedure accesses the Engine objects directly, using the Engine aggregate, to write external effects onto the Engine objects. Figure 4-5, page 29, shows such an executive level connecting procedure. The fragment reads the spark from the **Ignition** object and writes the spark value to the **Burner** object in the Engine system.

Next, the procedure **Engine_System.Update_Engine_System** is called to process the system level connections. When this operation is finished, the Engine system update is complete and the system is consistent with all its external effects.

4.6.2. Advantages of Executives

The implementation of executives described in this chapter follows the same model of connections used at the system level. Additionally, the executive has scheduling information in the form of an activity table which defines an order for processing its systems. Using the activity table, tuning of the simulator system by balancing the system processing across the frames of the cyclic executive is simplified.

```
with Flight_Executive_Connection_Manager;
with Flight_System_Names;

with Engine_System;
with Electrical_System;

package body Flight_Executive is

  procedure Update_Flight_Executive (Frame : in
                    Global_Types.Execution_Sequence) is

  begin
    for A_System in Flight_System_Names.Name_Of_A_Flight_System loop
      if Its_Time_To_Do (Frame) (A_System) then
        case A_System is

          when Flight_System_Names.Electrical =>
            Flight_Executive_Connection_Manager.
                Process_External_Connections_To_Electrical_System;
            Electrical_System.Update_Electrical_System;

          when Flight_System_Names.Engine =>
            Flight_Executive_Connection_Manager.
                Process_External_Connections_To_Engine_System;
            Engine_System.Update_Engine_System;
        end case;
      end if;
    end loop;

  end Update_Flight_Executive;

end Flight_Executive;
```

Figure 4-10: Flight Executive Example

Distributed processing can be handled easily by partitioning executives across the available processors. More discussion of this topic is in Section 6.1.

4.7. Advantages of the Architecture of the Paradigm

The two main design goals for the paradigm were to eliminate unnecessarily layered objects and to simplify dependencies among objects. Both goals have been met.

The solution does not contain nested objects and the software architecture is flat. Connecting procedures provide the means of accessing objects for transferring state information. The connections at the executive level can access all objects, in the systems under the scope of the executive. Objects are accessed by name through the data structures which aggregate objects for each system. A procedure call is not required to obtain a "pointer" to the object being accessed. We assert that the solution is natural. A spark goes to a burner, not to an engine.

The abstraction of higher-level objects, such as engines, is captured in the notion of a system: a set of objects updated as an entity. The benefits of nested objects are retained, i.e., high-

level objects can be updated and reused as a single entity. This abstraction coupled with the approach to processing connections facilitates multiprocessing. Placing a set of systems on a separate processor requires only creating an executive for the processor and making minor changes to the executive level connections for the processor. None of the system level code changes.

The major difference between this paradigm and other object-oriented paradigms is the use of connecting procedures to transfer information. Connecting procedures allow objects and systems to stand-alone. Each can be developed independently. Connecting procedures provide a firewall: changes in implementation to objects on one side of a connection do not affect the implementation of objects on the other side.

Connecting procedures facilitate both independent development and reuse. In particular, connecting procedures provide a systematic way to handle typing mismatches. It is desirable, but not always possible, for two connected objects to use the same types to communicate.

The software partitioning of connecting procedures simplifies compilation dependencies. All access to objects happens through connecting procedures. Thus, it is only the procedures managing connections to a system that need to be recompiled if an object manager specification changes. Each of these connecting procedures can be implemented as a separate procedure; in the Ada sense.

Connecting procedures provide a consistent means of updating systems and objects. Thus, connecting procedures provide a means for specifying control flow. No extraneous concepts or operations are required.

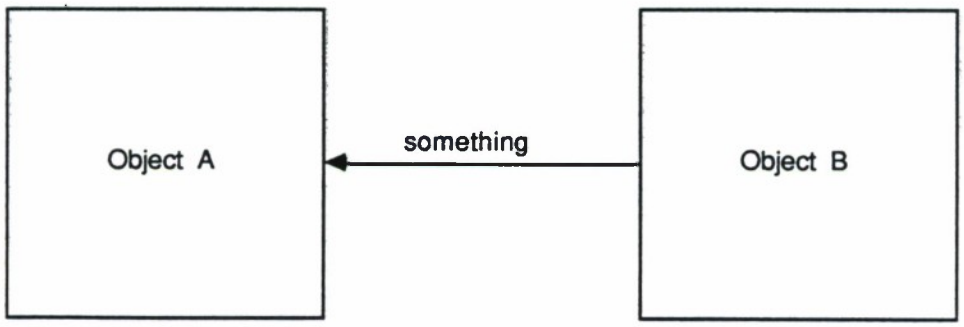


Figure 4-11: Object Dependency Example

Figure 4-11 illustrates some of the flexibility of connections.[28] Object B provides object A

[28]Figure 4-11 is the same as Figure 2-1.

with *something*, i.e., a connection exists, as shown, between A and B. Assume that A and B are in the same system.

1. If B needs, or is coded with, a different type of *something* than A, then the connection procedure converts the type.
2. If B moves to a different system, then the ownership of the connection is changed (from system level to executive level).
3. If B moves to a different processor, then connect the tail of the connection arrow to a "buffer" representing the other processor (See Section 6.1 for more information.)
4. If B needs to be stubbed, then the connecting procedure can be used as the stub.[29]

The paradigm produces software that is easy to modify. Typical modifications include adjusting the distribution of processing among the frames of a cyclic executive, adding malfunctions, adding or removing objects, and modeling wear and aging of components. Examples of some of the potential modifications are:

1. Moving the update of a system to a different frame requires a change only in the executive's schedule table
2. Adjusting the air flow, for one of the systems which uses air flow, can be done in a connecting procedure without worrying about side-effects in the other systems
3. Adding a malfunction to an engine component, the **Burner** for example, requires only the following:
    a. making the malfunction selectable at the Instructor Operator Station (IOS)
    b. adding a connection from the IOS buffer to the **Burner**
    c. changing the model of the **Burner**.
4. Adding a third compressor stage to the engine will not disturb the major math models of the engine. It requires only creating the object in software and adding connections to the object from the **Engine Casing** object
5. Modeling wear on a **Rotor** bearing requires adding the interface **Time_Has_Passed (Amount: Time)** to the object, making a small change to the private type, and reducing the efficiency of the **Rotor** in proportion to its time in service
6. Adding a system to an executive requires creating the system, its connection and aggregate packages, and the objects necessary to describe the system. The system's objects need to be accessible by the executive's connection package, see Figure 3-5, and the system itself needs to be included in the schedule table and update procedure of the executive. The software architecture tends to grow out (or flat) not down.

[29]This technique is used consistently throughout the Engine code in Appendix C. The examples used in this chapter, for example Figure 4-5 which shows a connection to the **Ignition** object manager, were constructed for illustrative purposes only.

5. Development Process

5.1. Role of the Paradigm

The development of systems using the paradigm is a design activity. The paradigm molds the designer's analysis of the requirements. The paradigm accommodates objects and connections. The result of the analysis of the requirements is a set of real-world objects and connections grouped into systems. Once this choice is made, the paradigm dictates the implementation.

The paradigm can be viewed as a means of consistently specifying objects, connections, systems, and executives. The result is a consistent implementation. Maintainers do not need to learn the architecture of each system. If the paradigm is followed, all systems will look the same.

During acquisition, the architecture of each system does not need to be evaluated. The quality of the architecture that follows from the paradigm needs to be evaluated only once. Design reviews can focus on the analysis of requirements, the choice of objects and connections, and system groupings.

5.2. Templates and Reuse

The object diagram, Figure 4-2, used four icons to describe the Engine system. Objects are represented by rectangles, connections between objects by arrowed lines, roundtangles are systems, and executives are defined by an irregular shape outlined in gray. Software systems of flight simulators can be defined in terms of these four icons.

The software architecture, Figure 3-5, can be derived mechanically from the object diagram. Each of the four icons is associated with its own set of one or more software package templates. Relationships among the software package templates are homomorphic to the relationships among the icons of the object diagram.[30]

[30] *Homomorphic* means "the same in meaning yet shown with a different structure".

```
with Standard_Engineering_Types;
package <Object>_Object_Manager is

   package Set renames Standard_Engineering_Types;
   type <Object> is private ;
   type <Attribute_2> is ??;
   type <Attribute_1> is ??;

   function New_<Object> return <Object>;
   --|*****************************************************************
   --| Description:
   --|    This function returns a pointer to a new <object> object
   --|    representation. This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return <object> which is an access to a <object> object.
   --|*****************************************************************

   procedure Give_<State_1>_To (A_<Object> : in <Object>;
                    Given_<Input>_<Type_1> : in Set.<Type_1>;
                    Given_<Input>_<Type_2> : in Set.<Type_2>;
                    Given_<Input>_<Type_3> : in Set.<Type_3>);
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified <object> object's
   --|    state given the <input>_<type_1>, <input>_<type_2>,
   --|    and the <input>_<type_3>.
   --|
   --| Parameter Description:
   --|    A_<object> identifies the <object> whose state is to be changed.
   --|    Given_<input>_<type_1> is the <input> <type_1>, in ??units
   --|    Given_<input>_<type_2> is the <input> <type_2>, in ??units
   --|    Given_<input>_<type_3> is the <input> air flow, in ??units
   --|*****************************************************************

   pragma Inline (Give_<State_1>_To);

private
   type <Object>_Representation;
   -- incomplete type, defined in package body

   type <Object> is access <Object>_Representation;
   -- pointer to an <object> representation

end <Object>_Object_Manager;
```

Figure 5-1: Object Manager Template Example

The templates contain the general features of the component, with place-holders for the specific features. Appendix B contains a complete object template. The template uses the notation **<object>** as a place-holder for the name of the object. The notation **<attribute_x>** is used for expression of operational state variables and attributes. The object operations are expressed in similar terms (See Figure 5-1).

The templates are not intended to contain all the necessary details for generating a complete version of the code. They are intended as a starting point. The framework for each object manager, system update package, connection package, and system aggregate is similar. The

details are different. Package bodies and subprogram bodies are provided within the templates. The implementor provides details within a template's framework. The resulting components will have a similar look and structure. This will aid readability, understanding, and maintenance.

5.2.1. Diagram Parsers

Several commercial tools have the capability of parsing diagrams and generating code templates to varying levels of detail. The detail is limited by the diagram notation.

The object diagram, Figure 4-2, is typical of a diagram for which a parser could be written. The parser could generate the templates discussed earlier. We view this as a natural extension of the paradigm toward a more automated solution.

5.3. Enhancements to Object Diagrams

The notation used on the object diagram, Figure 4-2, reflects the dependencies between objects and state information. It defines the connections necessary to construct the system.

Several extensions to the diagram notation can be envisaged. One would be to delineate the processing order of the connections. The **Engine Casing** is the object through which the air flows as the air passes through the engine. Each of the other objects interacts with the air as it flows through the **Engine Casing**. Nothing on the diagram denotes the order of connection processing. There may, however, be a specific order necessary to insure a consistent state of the Engine system.

Another extension would be to add pointers to algorithms. The algorithms, expressed in pseudocode, could be inserted in package bodies by the diagram parser.

6. Open Issues

6.1. Distributed Processing

One of the design goals of the paradigm was to facilitate spreading the work load over multiple processors. The description that follows encompasses our theories on what would be required to distribute the processing over several processors. We have not implemented or tested any of these ideas.

The paradigm begins with the notion of an executive. An executive controls the update of a set of systems compiled together and thus running on a single processor. The paradigm assumes that there will be more than one set of systems and that multiprocessing will be involved.

```
Integrated_Drive_Energy :=
  Generator_Object_Manager.Get_Energy_From
    (A_Generator => Ac_Power_Aggregate.Integrated_Drive_Generators
                    (Engines_To_Idg_Map (An_Engine)));

Rotor2_Object_Manager.Give_Torque_To
  (A_Rotor2    => Engine_System_Aggregate.Engines
                    (An_Engine).The_Rotor2,
   Given_Torque => Standard_Engineering_Types.Torque
                    (Integrated_Drive_Energy));
```

Figure 6-1: Executive Connection Procedure Example

```
Integrated_Drive_Energy :=
  Flight_Executive_Buffer.Get_Energy_From
    (A_Buffer_Location => Flight_Buffer.Idg(An_Engine);

Rotor2_Object_Manager.Give_Torque_To
  (A_Rotor2    => Engine_System_Aggregate.Engines
                    (An_Engine).The_Rotor2,
   Given_Torque => Standard_Engineering_Types.Torque
                    (Integrated_Drive_Energy));
```

Figure 6-2: Communicating with a Data Transfer Buffer

The abstraction of higher-level objects, such as engines, into systems allows a set of objects to be updated as an entity. This abstraction coupled with the paradigm's approach to processing connections facilitates multiprocessing. Placing a set of systems on a separate processor requires only creating an executive for the processor and making minor changes to the executive-level connections to the system.[31] None of the system-level code changes.[32]

Communication between executives is handled by an abstraction called a *buffer*. A *buffer* is some means of sharing data among separately compiled software.[33] The paradigm makes no assumption about how the operating system transfers data or how executives on separate processors are invoked. For example, assume that the Flight Executive has been split so that some of its systems, e.g., the Electrical system and the Fuel system, are on a processor separate from the Engine system. The executive that handles the Engine system needs to communicate with a buffer to get the environmental effects from these other systems. Figure 6-1 shows how connections between objects are typically handled. Figure 6-2 shows how communication between the executive's connecting procedure and a buffer can be implemented. The fragments read the torque required by the **Integrated Drive Generator** object manager in the Electrical system from the buffer. This is one of the changes required to implement a system on distributed processors.

Another required change would be to load the buffer with the states of objects needed by systems on the other processor. All outputs required by systems on other processors must be written into the buffer. This step would take place after the update of the system, as defined in Section 4.5.2.

One can imagine a development environment which automatically accommodates the distribution of systems across processors. The notations for the object diagram could be extended to indicate which systems were to be grouped on a processor. The "address" of the object read by a connection procedure could be calculated at link time: the choices would be an object or a buffer surrogate.

6.2. Tuning

The construction of a system using the paradigm results in a product which is easy to read, understand, and maintain. The performance of the system, however, still must fit into the time constraints demanded by the application. The implementation described in the paradigm (and embodied in the templates) is intended to be a starting point for a usable system.

---

[31]A typical minor change is demonstrated in Figures 6-1 and 6-2.

[32]There are many approaches to the solution of this problem. We do not intend to compare or delineate all possible solutions. One other solution would be to have the generation of connection dependencies handled by compiler pragmas. The effects would be the same, however. Our goal was to minimize perturbations to the connection procedures.

[33]In our observations of flight simulators, a *buffer* is a record data structure used in the communication between processors.

We fully expect that adjustment of some of the concepts may be necessary. For example, Ada allows an implementor to *inline* certain procedures and functions. The overhead of a procedure call is saved. For many of the object manager operations, which are only a few lines long and tend to be called frequently during an update, *inlining* may provide a significant time savings.

Another useful technique is that of combining effects. For example, providing multiple parameters to a subprogram instead of making multiple, individual subprogram calls. The implementation of the Engine system, described in Chapter 4, demonstrates this technique. Figure 4-6 shows an example with three parameters in the subprogram call **Get_Discharge_Air_From**.

A second method for combining effects is to group like objects together. For example, in a simulator electrical system there are hundreds of circuit breakers. Each one has to be updated with respect to the hardware linkage buffer on each cycle. Also, at each level several breakers have to be updated through their connections to other systems. One solution is to create an object manager that handles groups of identical objects. A circuit breaker collection manager would contain subprograms for dealing with groups of breakers at a time. Thus, a single subprogram call operating on a group of objects replaces multiple calls each operating on individual objects.

6.3. Reposition and Flight Freeze

Flight freeze and reposition are two of the operating modes of an aircraft simulator.

In the flight freeze mode the simulator software state is frozen, i.e., it stops changing with time. Communication with the simulator hardware must be maintained. Freeze may be initiated by the instructor at any time during a training exercise when communication with students is necessary.

The reposition mode is initiated by the instructor at the IOS when a particular training exercise is to be repeated. The communication between the simulator software and the hardware is maintained, and new values for flight data are loaded into the software. After a sufficient waiting period to allow the software to ramp to the new conditions, the simulator is restarted.

The paradigm considers time to be an outside effect on an object. Thus, it might be possible to implement flight freezes by controlling the time effects on objects. Similarly, a reposition would be accomplished by using reposition connecting procedures. In reposition mode, the executive level would connect systems to reposition buffers. A connecting procedure would read from the buffer instead of the object it reads from during normal run mode.

We have not implemented or tested these ideas. However, we are convinced that the paradigm does not complicate reposition and flight freeze.

6.4. System Exports and System Imports

The paradigm makes the objects of a system visible to the executive and to other systems.[34] We have begun to study the merits of hiding the objects behind system-level data called export and import areas.

Currently, the paradigm calls for a connection to read the state of one object and write the state of another object. If export and import areas are used, executive-level connections would move data from the export areas of some systems to the import areas of others.

For example, the Engine system's import area would contain a typed variable for each state of its objects affected by other systems. Before calling for an update of the Engine system, the executive would process the connections to the Engine's import area. The Engine system would have system-level connections from the import area to its objects. To perform an update, the Engine system would gate those connections in conjunction with the connections among the Engine objects. Upon completion of the update, the Engine system would gate connections from its objects to its export area. Figure 6-3 shows how the Engine object diagram would appear using export and import areas. Figure 6-4 shows how the software architecture would appear.[35]

We have not completed analysis of this approach. The only apparent benefit is subtle. Remember that an object can update its state when given a new environmental effect value. One usually assumes that moving data is a fast operation. However, if the executive writes directly to an object, which updates its state, then the movement of data could be time consuming. The use of export and import areas assumes that all computations will occur during the update of the system instead of before or after the update is called for by the executive.

The use of export and import areas does not hide information. The executive-level connections will terminate either at objects or at export and import areas. In either case the executive knows about the same number of reads and writes. The important thing is that even without export and import areas, a system knows nothing about any other systems.

6.5. Our Executive's Control of Time

A system expects to be updated at regular intervals. It also expects to update itself with respect to a consistent set of external effects. Suppose the Engine system was last updated at *t20*. The executive is preparing to update the Engine at *t25*. The Fuel system has been updated at *t25*, but the Electrical system was last updated at *t20* and will not be updated again until *t30*.

---

[34]However, states are changed and read only through connections. Thus, no system reads from or writes to other systems.

[35]Compare Figures 6-3 and 6-4 with Figures 4-2 and 3-5, respectively.

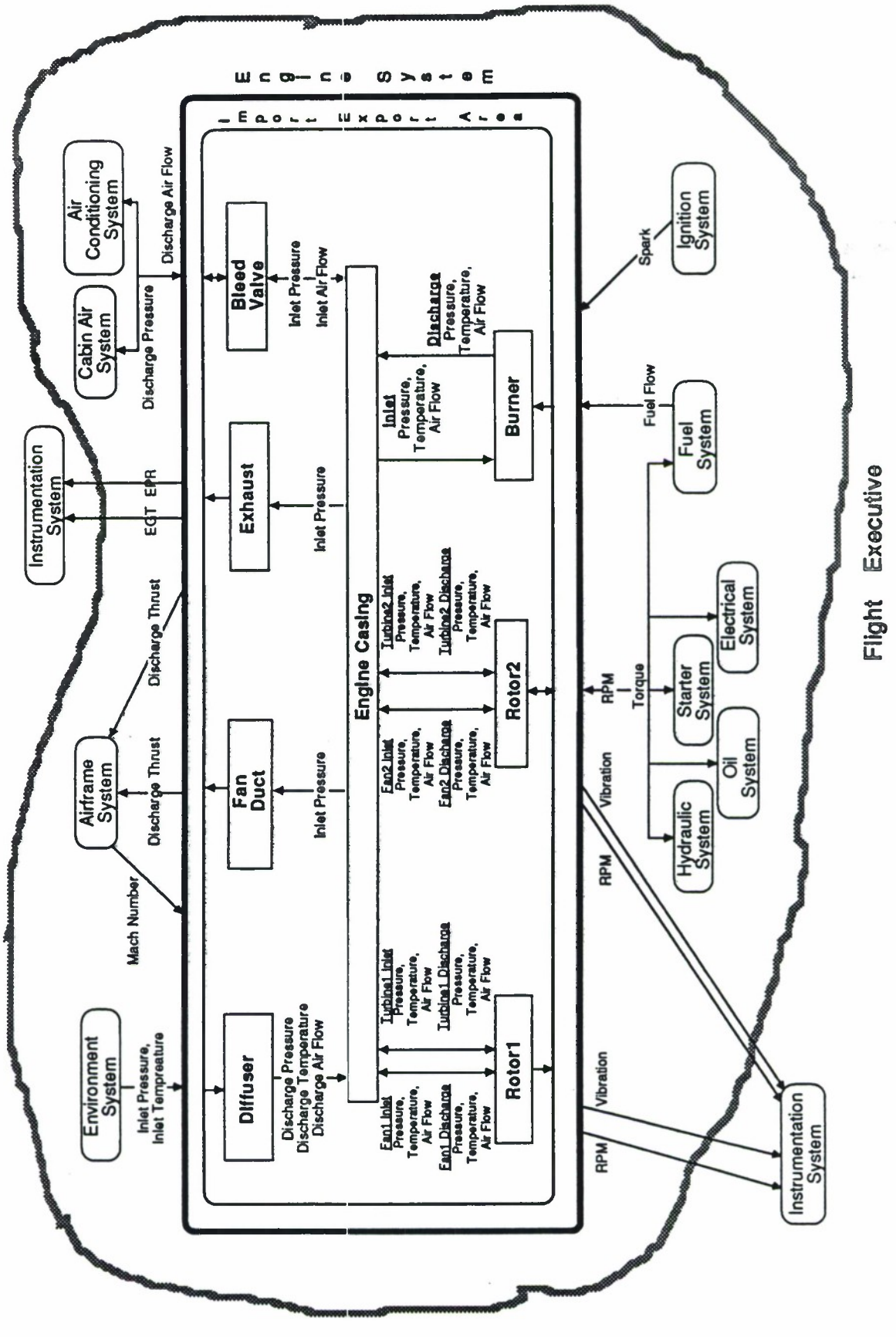


Figure 6-3: Alternative Engine Object Diagram

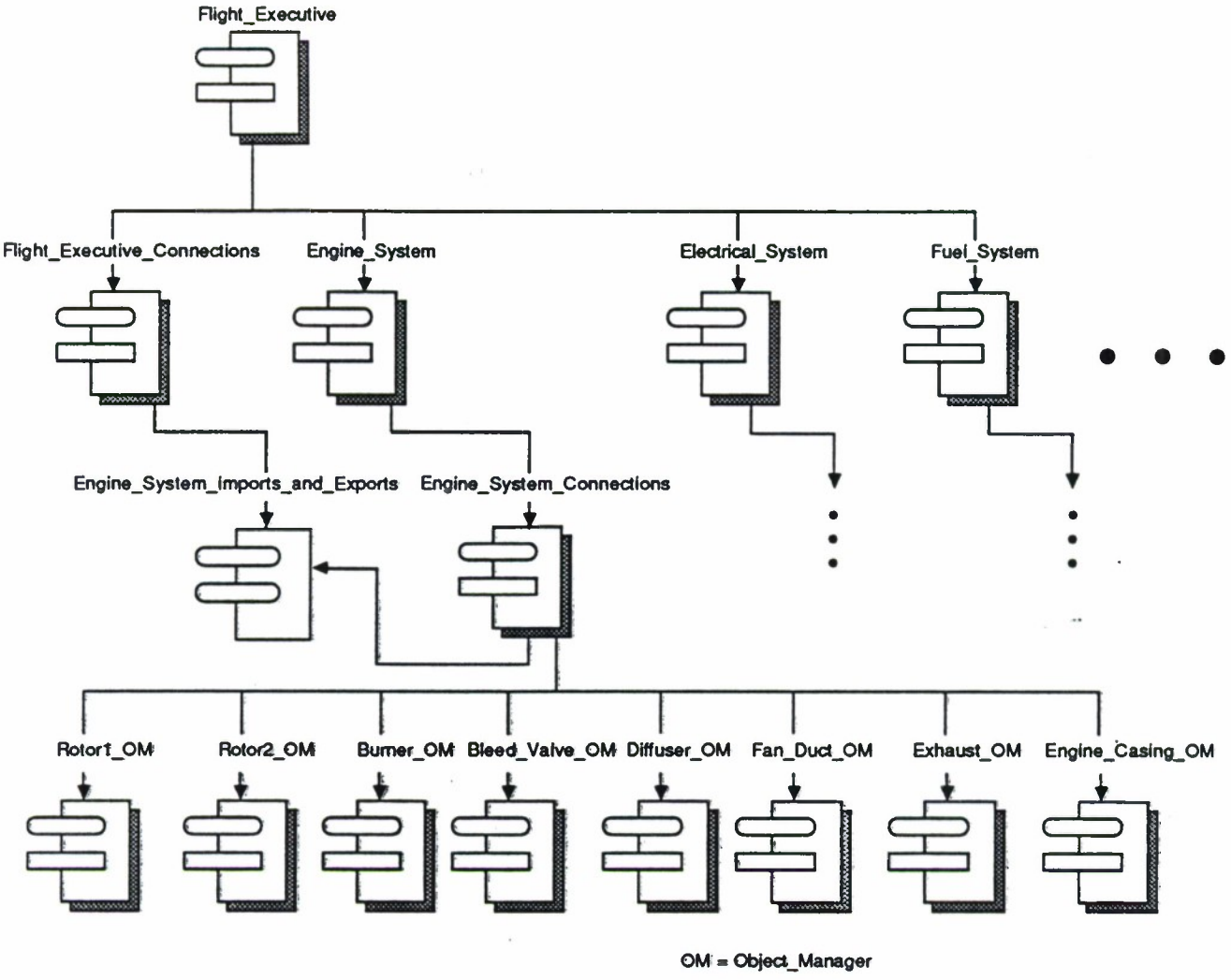


Figure 6-4: Alternative Software Architecture

To present the Engine with a consistent picture of its world at *t25*, the executive must extrapolate the state of the Electrical system. The executive must keep a history of snapshots, in this case one from *t10* and one from *t20*. We have not incorporated this in the paradigm, but we see no problem in doing so.

6.6. Cyclicness

We have not studied the extent to which the executive must be cyclic. However, we see no reason to force updates to be harmonic. An executive is nothing more than a dispatcher that controls time and access to a central resource, the CPU. Algorithms similar to the rate monotonic algorithms, becoming available for scheduling tasks, could be used to schedule the updates of systems. Since the algorithms will allow tasks to run at discordant periods, the executive would not have to be cyclic in the traditional sense.

6.7. Load Balancing

```
with Flight_System_Names;
with Flight_Executive_Connection_Manager;

with Engine_System;
with Electrical_System;

package body Flight_Executive is

  procedure Update_Flight_Executive (Frame : in
                    Global_Types.Execution_Sequence) is
  begin
    for A_System in Flight_System_Names.Name_Of_A_Flight_System loop
      if Its_Time_To_Do (Frame) (A_System) then
        case A_System is

          when Flight_System_Names.Electrical =>
            Flight_Executive_Connection_Manager.
              Process_External_Connections_To_Electrical_System;
            Electrical_System.Update_Electrical_System;

          when Flight_System_Names.Engines_First_Part ..
                          Flight_System_Names.Engines_Fourth_Part =>
            if A_System = Flight_System_Names.
                                        Engines_First_Part then
                        Flight_Executive_Connection_Manager.
                              Process_External_Connections_To_Engine_System;
                      end if;
            Engine_System.Update_Engine_System (A_System);
        end case;
      end if;
    end loop;

  end Update_Flight_Executive;
end Flight_Executive;
```

Figure 6-5: Executive Example

The paradigm, as described, requires an update to be atomic. This means that when the executive regains the CPU after calling a system update, the executive expects all the objects of the system to be consistent with respect to the external world presented to the system at the start of the update. However, it may not be possible to schedule the update of an entire system in a single time slice.

The paradigm can be modified to accommodate spreading an update across frames of a cyclic executive. For example, it may be necessary to distribute the update of the Engine system equally to four frames. In this case, the executive would process all connections to the Engine system before calling for the update of the Engine system. The executive would then call the Engine update procedure four times. The update procedure would include a parameter indicating the cardinality of the update: first, second, third, or fourth. The system would know what had to be done at each call. Figures 6-5 and 6-6 show how the executive-level and system-level routines would be modified to accommodate this concept.[36]

[36]Compare Figures 6-5 and 6-6 to Figures 4-10 and 4-6, respectively.

```
with Flight_System_Names;
with Engine_System_Aggregate;

with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;

package body Engine_System is

  procedure Update_Engine_System (
                A_System: Flight_System_Names.Name_Of_A_Flight_System)is

    Diffuser_Discharge_Pressure   : Set.Pressure;
    Diffuser_Discharge_Temperature : Set.Temperature;
    Diffuser_Discharge_Air_Flow   : Set.Air_Flow;

  begin
       case A_System is
         when Flight_System_Names.Engines_First_Part =>
           --
           -- Model the connection characterized by the dependence of the
           -- Engine Casing on the Diffuser for Pneumatic_Energy.
           --
               Diffuser_Object_Manager.Get_Discharge_Air_From
                    (A_Diffuser =>
                         Engine_System_Aggregate.Engines
                              (Given_Engine_Name).The_Diffuser,
                    Returning_Discharge_Pressure =>
                         Diffuser_Discharge_Pressure,
                    Returning_Discharge_Temperature =>
                         Diffuser_Discharge_Temperature,
                    Returning_Discharge_Air_Flow =>
                         Diffuser_Discharge_Air_Flow);

               Engine_Casing_Object_Manager.Give_Air_Flow_To
                    (A_Engine_Casing =>
                         Engine_System_Aggregate.Engines
                              (Given_Engine_Name).The_Engine_Casing,
                         Given_Air_Flow => Diffuser_Discharge_Air_Flow);
                         Process_First_Connection_Set;

         when Flight_System_Names.Engines_Second_Part => null;
           --
           -- do some other Engine connections here
           --

         when Flight_System_Names.Engines_Third_Part => null;
           --
           -- do still more Engine connections here
           --

         when Flight_System_Names.Engines_Fourth_Part => null;
           --
           -- do the rest of the Engine connections here
           --
       end case;
  end Update_Engine_System;
end Engine_System;
```

Figure 6-6: System Example

Note that the executive would not know how the Engine system distributed its work. The

executive assumes that between the first and fourth call of the Engine update the state of the Engine system is undefined. This is not a hardship; the executive merely broadcasts Engine state, i.e., gates the connections from the Engine system, to other systems before the update begins.

6.8. Generics

Each object manager in the paradigm provides an allocator, the **New_<object>** operation, to create instances of an object. For example, there are four instances of the **Burner** object, one for each engine.

Reviewers have suggested generics as an alternative approach for creating multiple instances of an object. Each object manager could be a generic package which would be instantiated once for each object needed. The generic parameters could be values for attributes and the operating conditions of the object. Figure 6-7 shows a generic **Burner** object manager.[37] In this case, there are no generic parameters.

There are two immediate advantages to such an approach. First, the aggregate data structure would not be needed. The instances would be named through instantiation. Figure 6-8 shows how the generic object managers would be instantiated for the Engine system.[38]

Second, memory for the objects would be allocated statically. The current implementation causes memory to be allocated on a heap, even though the number of instances of each object is constant.

The use of generics to create systems is more complicated. Executive-level connections to systems do not flow between the same instances of objects. We are continuing to investigate the use of generics at this level.

6.9. System-Level Objects

In the paradigm, the Engine system is the sum of its parts. There is no system-level, Engine system object per se. The Engine system has no state other than the states of the objects which make up the Engine system.

Some systems seem to require a system-level object. For example, the case of an IFF[39] box might distribute power to the components of the IFF system inside the case. Under the paradigm, software objects would be created to model the selected components. The case itself also might be modeled with a software object. The software object corresponding to the

---

[37]Compare with the **Burner** object manager in Figure 4-3.

[38]Compare with the Engine Aggregate package shown in Figure 4-8.

[39]Identify Friend or Foe

```
with Standard_Engineering_Types;

generic

package Burner_Object_Manager is

   package Set renames Standard_Engineering_Types;

  type Burner is private ;
  -- an Burner is an abstraction of a Burner within an Engine.

  type Spark is (None, Low, High);
  -- burner needs only to know relative spark size

  type Fuel_Flow is (None, Flowing);
  -- the burner needs to know only if it has fuel available

  function New_Burner return Burner;

  procedure Give_Inlet_Air_To
        (A_Burner           : in Burner;
         Given_Inlet_Pressure   : in Set.Pressure;
         Given_Inlet_Temperature : in Set.Temperature;
         Given_Inlet_Air_Flow   : in Set.Air_Flow);

  procedure Get_Discharge_Air_From
                (A_Burner : in Burner;
                 Returning_Discharge_Pressure   : out Set.Pressure;
                 Returning_Discharge_Temperature : out Set.Temperature;
                 Returning_Discharge_Air_Flow   : out Set.Air_Flow);

  procedure Give_Fuel_Flow_To
                (A_Burner      : in Burner;
                 Given_Fuel_Flow : in Fuel_Flow);

  procedure Give_Spark_To (A_Burner   : in Burner;
                           Given_Spark : in Spark);

pragma Inline (Give_Inlet_Air_To,
               Get_Discharge_Air_From,
               Give_Fuel_Flow_To,
               Give_Spark_To)

private
  type Burner_Representation;
  -- incomplete type, defined in package body

  type Burner is access Burner_Representation;
  -- pointer to an Burner representation

end Burner_Object_Manager;
```

Figure 6-7: Generic Object Manager Example

case would be an object at the same level as the objects inside the case. We would not feel compelled to nest component software inside the case software.

We will continue to investigate the need for objects which aggregate system-level information.

```
with Burner_Object_Manager;
package Engine_1_Burner is new Burner_Object_Manager;
with Burner_Object_Manager;
package Engine_2_Burner is new Burner_Object_Manager;

with Bleed_Valve_Object_Manager;
package Engine_1_Bleed_Valve is new Bleed_Valve_Object_Manager;
with Bleed_Valve_Object_Manager;
package Engine_2_Bleed_Valve is new Bleed_Valve_Object_Manager;

with Diffuser_Object_Manager;
package Engine_1_Diffuser is new Diffuser_Object_Manager;
with Diffuser_Object_Manager;
package Engine_2_Diffuser is new Diffuser_Object_Manager;

with Engine_Casing_Object_Manager;
package Engine_1_Engine_Casing is new Engine_Casing_Object_Manager;
with Engine_Casing_Object_Manager;
package Engine_2_Engine_Casing is new Engine_Casing_Object_Manager;

with Exhaust_Object_Manager;
package Engine_1_Exhaust is new Exhaust_Object_Manager;
with Exhaust_Object_Manager;
package Engine_2_Exhaust is new Exhaust_Object_Manager;

with Fan_Duct_Object_Manager;
package Engine_1_Fan_Duct is new Fan_Duct_Object_Manager;
with Fan_Duct_Object_Manager;
package Engine_2_Fan_Duct is new Fan_Duct_Object_Manager;

with Rotor1_Object_Manager;
package Engine_1_Rotor1 is new Rotor1_Object_Manager;
with Rotor1_Object_Manager;
package Engine_2_Rotor1 is new Rotor1_Object_Manager;

with Rotor2_Object_Manager;
package Engine_1_Rotor2 is new Rotor2_Object_Manager;
with Rotor2_Object_Manager;
package Engine_2_Rotor2 is new Rotor2_Object_Manager;
```

Figure 6-8: Generic Object Instantiation Example

7. Electrical System

An Electrical system in an aircraft provides electrical power to devices in other systems: devices such as fuel pumps and valves in the Fuel system, hydraulic pumps in the Hydraulic system, and air conditioning in the Environmental Control system. The systems are able to function only if power is available. They, in turn, put their load, i.e., the amount of current they require, back onto the Electrical system. The load is transferred back to the generators, along the Electrical system buses, where a determination of possible overloading takes place.

A subset of the Electrical system has been completed and tested. The code with accompanying documentation is available on request from the authors. The code illustrates some concepts not illustrated by the Engine system example.

7.1. Additional Concepts

The Engine system, Appendix C, is complete through the package specifications. The subset of the Electrical system is fully functional and has been thoroughly tested.

Several performance issues arose during the implementation. There are several hundred **Circuit Breakers** in a typical Electrical system. Each one has to be updated with respect to the hardware linkage buffer on each cycle. Also, at each level of the software several breakers have to updated through their connections to other systems. The subprogram calls in each object manager were *inlined* in order to reduce the overhead during update.

Grouping of like effects is also performed. Voltage and load conversion factor (lcf) are updated together. In addition, voltage, lcf, and current are grouped in a data structure which is used during all read operations from objects. Both steps result in fewer subprogram calls.

The concept of updating a system as a unit means, to us, that all aspects of the system update must be complete in the execution frame. The Electrical system subset includes a tie-bar, an electrical bus which connects several other buses. In order to insure that the update is complete within the frame, the tie-bar is processed repeatedly in the frame. The number of times necessary depends on the number of other connections to the tie-bar.

Other issues that arose during the complete implementation included decisions about writ-

ing effects to objects and reading outputs from objects. For some objects, like **Circuit Breakers**, the external effects are written and outputs are calculated during a read operation. For other objects, states are calculated when effects change.

The Electrical system object diagrams look like circuit diagrams. Given a library of objects and a diagram parser, one could fully automate the production of code from a circuit diagram.

Appendix A: Software Architecture Notation

The notation used to describe the software architecture is a modified form of the notation expounded by Grady Booch in his book on software engineering with Ada [1] and his book on reusable software components with Ada [2]. The notation used is true to the intent of Booch's notation. The variations (i.e., extensions) are:

- use of reduced package, subprogram and task icons inside larger icons rather than the object (or blob) icon
- use of object dependency arrows more subtly, to distinguish different types of dependencies
- internal details of any reusable subsystem, package, subprogram or task are not shown

One final note about the notation: The figures need not show all the fine-grained detail of a package or subprogram. When the code of a package (or subprogram) is compared to a figure associated with that package (or subprogram) there may be nested procedures or packages not shown on a particular picture, or it may depend on a package not explicitly shown in the figure. The guidelines for these cases are:

- utility packages or services are not shown (this includes things like text_io, reusable data structure packages, math libraries, etc.)
- the figures are meant to show the significant details at a particular level, not all the details
- the definition of "a significant detail" is solely at the discretion of the designer

Based on these ideas, Figures A-1 thru A-4 explain the meaning of each of the icons available using this notation.

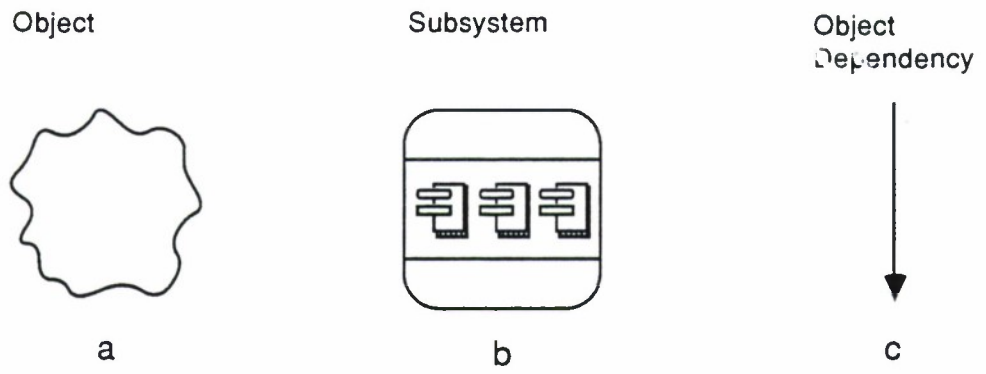


Figure A-1: Object, Subsystem and Dependency Notation

The object (or blob) icon, shown above in Figure A-1 (a), represents an identifiable segment of a system, about which we have no implementation information. We make no use of this icon.

The subsystem icon, shown above in Figure A-1 (b), represents a major system component that has a clearly definable interface, yet, which is not representable as a single Ada package. We currently make no use of this icon, although we could.

The object dependency symbol, shown above in Figure A-1 (c), indicates that the object at the origin of the arrow is dependent on the object at the head of the arrow. The origin of the arrow indicates where the dependency occurs. If the origin is in the white area of an icon (shown in subsequent figures), it indicates a specification dependency. If the origin is in the shaded area, it indicates a body dependency.

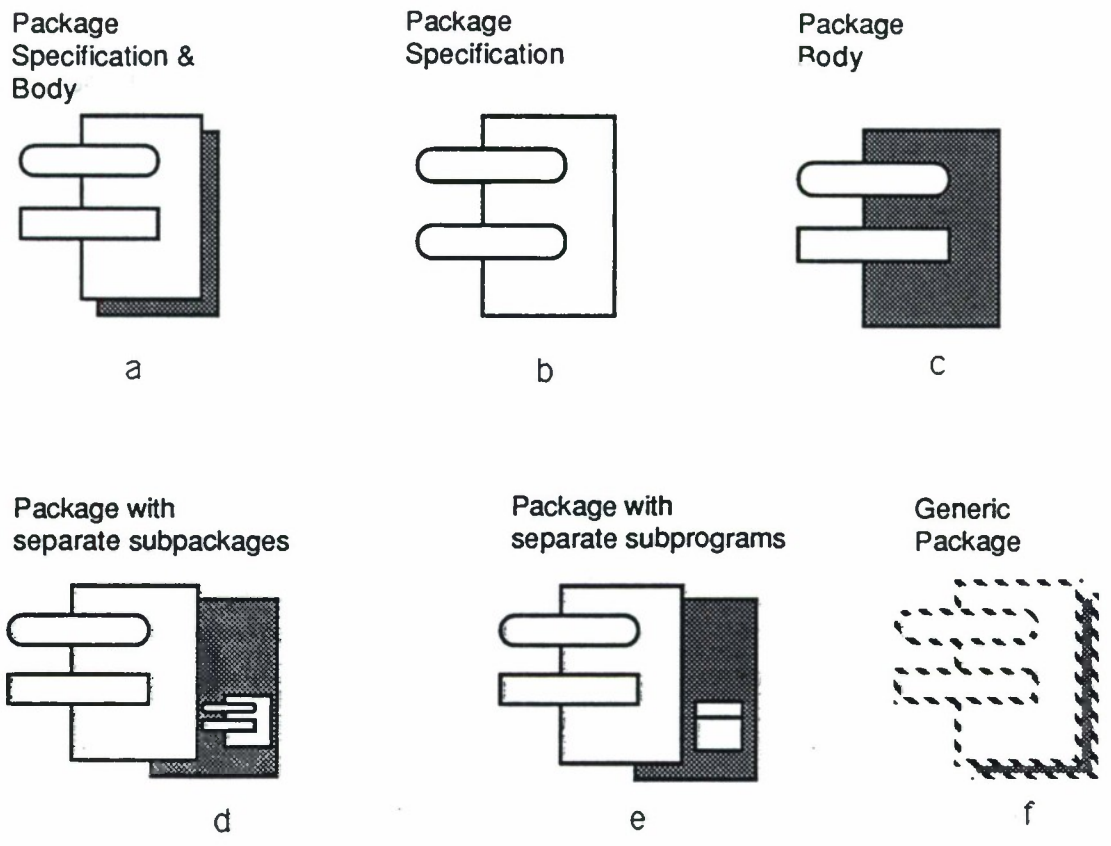


Figure A-2: Package Notation

The package specification and body icon, shown above in Figure A-2 (a), represents an Ada package specification, the white area, with an associated package body, the shaded area. This icon can be broken apart to show a package specification, Figure A-2 (b), or a package body, Figure A-2 (c).

Figures A-2 (d) and (e) are variations on the package icon which show greater detail. Figure A-2 (d) is used to represent packages which have nested subpackages within the body; if the small package icon were placed within the specification, it would indicate visible nested packages. Similarly, Figure A-2 (e) illustrates the notation used for separate subprograms within the body of a package.

Finally, Figure A-2 (f) illustrates the icon used for generic packages. Everything discussed above in regard to regular packages can also be applied to generic packages.

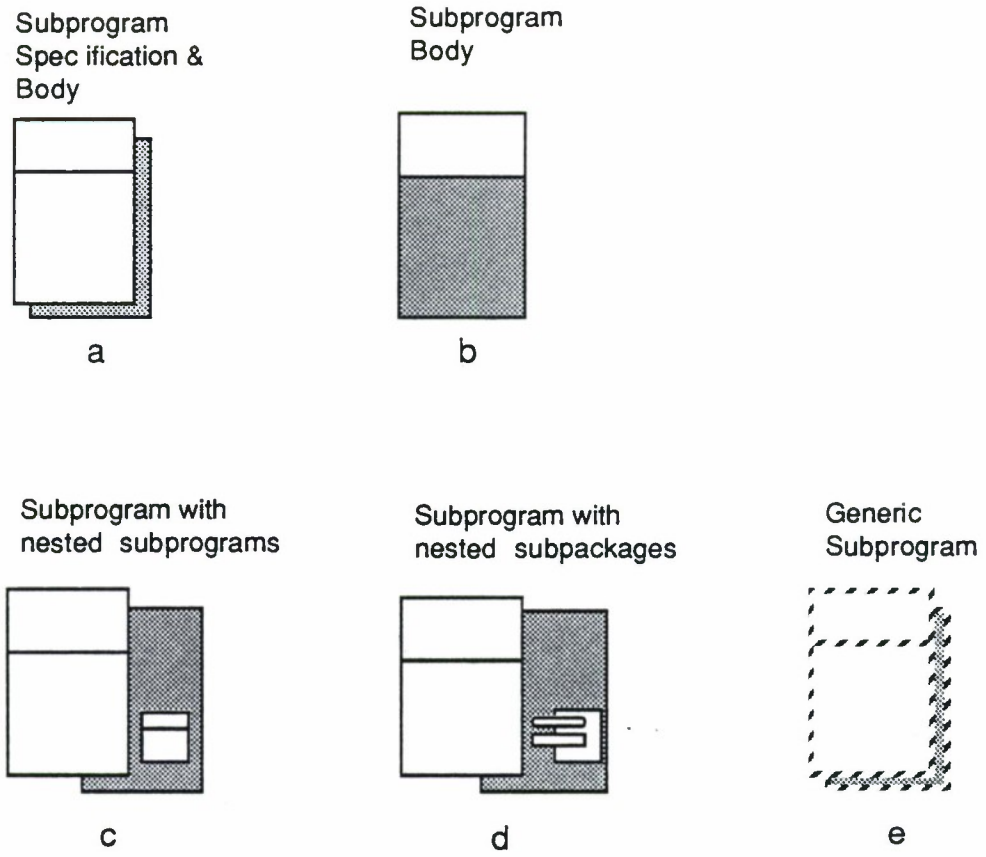


Figure A-3: Subprogram Notation

Much of what was discussed previously in regard to packages also applies to subprograms. The subprogram specification and body icon, shown above in Figure A-3 (a), represents an Ada subprogram specification, the white area, with an associated subprogram body, the shaded area. This icon can be broken apart to show a subprogram body, Figure A-3 (b).

Figures A-3 (c) and (d) are variations on the subprogram icon which show greater detail. Figure A-3 (c) is used to represent subprograms which have nested subprograms within the body. Similarly, Figure A-3 (d) illustrates the notation used for separate subpackages within the body of a subprogram.

Finally, Figure A-3 (f) illustrates the icon used for generic subprograms. Everything discussed above in regard to regular packages can also be applied to generic subprograms.

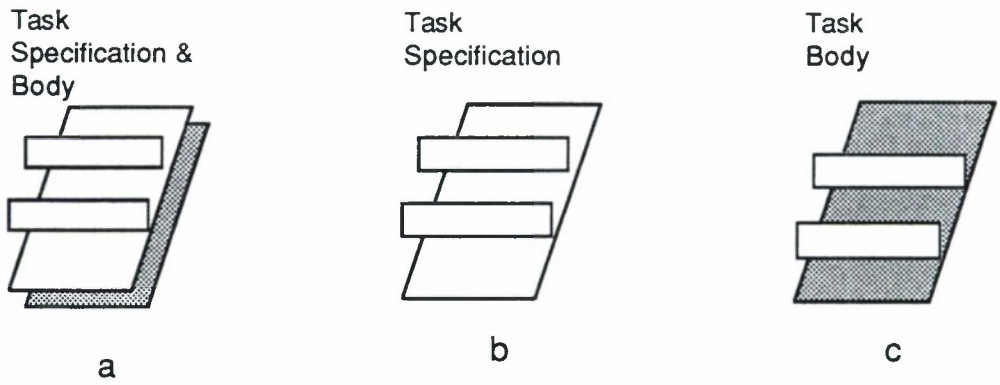


Figure A-4: Task Notation

Again, much of what was discussed previously in regard to packages and subprograms, applies to tasks. The task specification and body icon, shown above in Figure A-4 (a), represents an Ada task specification, the white area, with an associated task body, the shaded area. This icon can be broken apart to show a task specification, Figure A-2 (b), or a task body, Figure A-4 (c). Although they are not shown, nested package and subprograms are represented in exactly the same manner as shown in Figure A-2 for packages and subprograms.

Appendix B: Object Manager Template

```
-- The following are instructions regarding the use of this template.
-- We realize that the template does not encompass every procedure
-- which might be needed, however with alterations to the existing
-- procedures one can easily affect the neccessary changes.

  -- Do global substitutes on the following :
--    <object> gets the name of the object being created
--          ie. <object> => Burner
--
--    <input>  gets the general prefix for indicating that a
--      variable is input
--          ie.  <input> => Inlet
--
--    <output> gets the general prefix for indicating that a
--      variable is output
--          ie. <output> => Discharge
--
--    <state_n> is the state of the object you wish to modify
--       Note : n can take on values 1 to 3 for state_n.
--            If more states are needed the user should
--            cut the existing ones to create more.
--            ie. <state_1> => Air
--
--    <type_n> is the type of the variable within a state which
--      which you wish to modify.
--      Note : n can take on values 1 to 9 with 1-3 corresponding
--        to state_1 , ... The user should remove unwanted
--        type.
--        ie. <type_1> => Pressure
--
--    <attribute_n> are the attributes of an object you wish to modify.
--      Note : n can take on values 1 to 3
--          ie. <attribute_1> => spark
--
--    <type_n_units> and <attribute_n_units> are the units corresponding to
--      to the types and attributes used within the object manager.
--          ie. <type_1_units> => pounds per square inch
--          ie. <attribute_1_units> => joules
--
-- do a search now for all instances of ?? and fill in the
--   neccessary information

-- Finally the user should remove all unwanted code and comments


--|*****************************************************************
--| Module Name:
--|    <object>_Object_Manager
--|
```

```
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine <object> for the C-141 simulator.
--|    This management entails creation of <object> object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-------------------------------------------------------------------
--| Module Description:
--|    The <object> object manager provides a means to create
--|    an <object> object via the New_<object> operation and returns
--|    an identification for the <object>, which is to be used when
--|    updating/accessing the <object> object's state as described below.
--|
--|    The <object> object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_<input>_<state_1>_To
--|       2) Give_<input>_<state_2>_To
--|       3) Give_<input>_<state_3>_To
--|       4) Give_<attribute_1>_To
--|       5) Give_<attribute_2>_To
--|       6) Give_<attribute_3>_To
--|    operations, requiring the following external state information:
--|       1) <input>_<type_1>      <type_1_units>
--|          <input>_<type_2>      <type_2_units>
--|          <input>_<type_3>      <type_3_units>
--|       2) <input>_<type_4>      <type_4_units>
--|          <input>_<type_5>      <type_5_units>
--|          <input>_<type_6>      <type_6_units>
--|       3) <input>_<type_7>      <type_7_units>
--|          <input>_<type_8>      <type_8_units>
--|          <input>_<type_9>      <type_9_units>
--|       4) <attribute_1>       <attribute_1_units>
--|       5) <attribute_2>       <attribute_2_units>
--|       6) <attribute_3>       <attribute_3_units>
--|
--|    The <object> object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_<output>_<state_1>_From
--|       2) Get_<output>_<state_2>_From
--|       3) Get_<output>_<state_3>_From
--|    operations, yielding the following internal state information:
--|       1) <output>_<type_1>    <type_1_units>
--|          <output>_<type_2>    <type_2_units>
--|          <output>_<type_3>    <type_3_units>
--|       2) <output>_<type_4>    <type_4_units>
--|          <output>_<type_5>    <type_5_units>
--|          <output>_<type_6>    <type_6_units>
--|       3) <output>_<type_7>    <type_7_units>
--|          <output>_<type_8>    <type_8_units>
--|          <output>_<type_9>    <type_9_units>
--|
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--|  Testing and Validation:
--|    none
--|
```

```
--| Notes:
--|    none
--|
--|-------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--|  13Jul88    kl     Modified
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*************************************************************************
--|
with Standard_Engineering_Types;

package <Object>_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type <Object> is private ;
   -- an <object> is an abstraction of a <object> within an Engine.

   type <Attribute_1> is ??;
   -- the <object> needs to know ??

   type <Attribute_2> is ??;
   -- the <object> needs to know ??

   type <Attribute_3> is ??;
   -- the <object> needs to know ??

   function New_<Object> return <Object>;
   --|*********************************************************
   --| Description:
   --|    This function returns a pointer to a new <object> object
   --|    representation.  This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return <object> which is an access to a <object> object.
   --|*********************************************************


   procedure Give_<Input>_<State_1>_To (A_<Object> : in <Object>;
                     Given_<Input>_<Type_1> : in Set.<Type_1>;
                     Given_<Input>_<Type_2> : in Set.<Type_2>;
                     Given_<Input>_<Type_3> : in Set.<Type_3>);
   --|*********************************************************
   --| Description:
   --|    Initiates a change in the specified <object> object's
   --|    state given the <input>_<type_1>, <input>_<type_2>,
   --|    and the <input>_<type_3>.
   --|
   --| Parameter Description:
   --|    A_<object> identifies the <object> whose state is to be changed.
   --|    Given_<input>_<type_1> is the <input> <type_1>, in <type_1_units>
   --|    Given_<input>_<type_2> is the <input> <type_2>, in <type_2_units>
   --|    Given_<input>_<type_3> is the <input> <type_3>, in <type_3_units>
   --|*********************************************************
```

```
procedure Give_<Input>_<State_2>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_4> : in Set.<Type_4>;
                  Given_<Input>_<Type_5> : in Set.<Type_5>;
                  Given_<Input>_<Type_6> : in Set.<Type_6>);
--|*************************************************************
--| Description:
--|    Initiates a change in the specified <object> object's
--|    state given the <input>_<type_4>, <input>_<type_5>,
--|    and the <input>_<type_6>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is to be changed.
--|    Given_<input>_<type_4> is the <input> <type_4>, in <type_4_units>
--|    Given_<input>_<type_5> is the <input> <type_5>, in <type_5_units>
--|    Given_<input>_<type_6> is the <input> <type_6>, in <type_6_units>
--|*************************************************************

procedure Give_<Input>_<State_3>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_7> : in Set.<Type_7>;
                  Given_<Input>_<Type_8> : in Set.<Type_8>;
                  Given_<Input>_<Type_9> : in Set.<Type_9>);
--|*************************************************************
--| Description:
--|    Initiates a change in the specified <object> object's
--|    state given the <input>_<type_7>, <input>_<type_8>,
--|    and the <input>_<type_9>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is to be changed.
--|    Given_<input>_<type_7> is the <input> <type_7>, in <type_7_units>
--|    Given_<input>_<type_8> is the <input> <type_8>, in <type_8_units>
--|    Given_<input>_<type_9> is the <input> <type_9>, in <type_9_units>
--|*************************************************************


procedure Get_<Output>_<State_1>_From
       (A_<Object> : in <Object>;
        Returning_<Output>_<Type_1> : out Set.<Type_1>;
        Returning_<Output>_<Type_2> : out Set.<Type_2>;
        Returning_<Output>_<Type_3> : out Set.<Type_3>);
--|*************************************************************
--| Description:
--|    Initiates a report of the specified <object> object's
--|    state returning the <output>_<type_1>,
--|    <output>_<type_2>, and the <output>_<type_3>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is needed.
--|    Returning_<output>_<type_1> is the <output>_<type_1> portion
--|       of <object> object's state, in <type_1_units>
--|    Returning_<output>_<type_2> is the <output>_<type_2> portion
--|       of <object> object's state, in <type_2_units>
--|    Returning_<output>_<type_3> is the <output>_<type_3> portion
--|       of <object> object's state, in <type_3_units>
--|*************************************************************

procedure Get_<Output>_<State_2>_From
       (A_<Object> : in <Object>;
        Returning_<Output>_<Type_4> : out Set.<Type_4>;
        Returning_<Output>_<Type_5> : out Set.<Type_5>;
        Returning_<Output>_<Type_6> : out Set.<Type_6>);
--|*************************************************************
--| Description:
--|    Initiates a report of the specified <object> object's
--|    state returning the <output>_<type_4>,
--|    <output>_<type_5>, and the <output>_<type_6>.
```

```
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is needed.
--|    Returning_<output>_<type_4> is the <output>_<type_4> portion
--|       of <object> object's state, in <type_4_units>
--|    Returning_<output>_<type_5> is the <output>_<type_5> portion
--|       of <object> object's state, in <type_5_units>
--|    Returning_<output>_<type_6> is the <output>_<type_6> portion
--|       of <object> object's state, in <type_6_units>
--|*************************************************************

procedure Get_<Output>_<State_3>_From
        (A_<Object> : in <Object>;
         Returning_<Output>_<Type_7> : out Set.<Type_7>;
         Returning_<Output>_<Type_8> : out Set.<Type_8>;
         Returning_<Output>_<Type_9> : out Set.<Type_9>);
--|*************************************************************
--| Description:
--|    Initiates a report of the specified <object> object's
--|    state returning the <output>_<type_7>,
--|    <output>_<type_8>, and the <output>_<type_9>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is needed.
--|    Returning_<output>_<type_7> is the <output>_<type_7> portion
--|       of <object> object's state, in <type_7_units>
--|    Returning_<output>_<type_8> is the <output>_<type_8> portion
--|       of <object> object's state, in <type_8_units>
--|    Returning_<output>_<type_9> is the <output>_<type_9> portion
--|       of <object> object's state, in <type_9_units>
--|*************************************************************


procedure Give_<Attribute_1>_To (A_<Object> : in <Object>;
                  Given_<Attribute_1> : in <Attribute_1>);
--|*************************************************************
--| Description:
--|    Initiates a change in the specified <object> object's
--|    state given the <attribute_1>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is to be changed.
--|    Given_<attribute_1> is the <attribute_1>, in <attribute_1_units>
--|*************************************************************

procedure Give_<Attribute_2>_To (
       A_<Object> : in <Object>; Given_<Attribute_2> : in <Attribute_2>);
--|*************************************************************
--| Description:
--|    Initiates a change in the specified <object> object's
--|    state given the <attribute_2>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is to be changed.
--|    Given_<attribute_2> is the <attribute_2>, in <attribute_2_units>
--|*************************************************************

procedure Give_<Attribute_3>_To (A_<Object> : in <Object>;
                  Given_<Attribute_3> : in <Attribute_3>);
--|*************************************************************
--| Description:
--|    Initiates a change in the specified <object> object's
--|    state given the <attribute_3>.
--|
--| Parameter Description:
--|    A_<object> identifies the <object> whose state is to be changed.
```

```
   --|    Given_<attribute_3> is the <attribute_3>, in <attribute_3_units>
   --|*********************************************************************


   pragma Inline (Give_<Input>_<State_1>_To,
                  Get_<Output>_<State_1>_From,
                  Give_<Input>_<State_2>_To,
                  Get_<Output>_<State_2>_From,
                  Give_<Input>_<State_3>_To,
                  Get_<Output>_<State_3>_From,
                  Give_<Attribute_1>_To,
                  Give_<Attribute_2>_To,
                  Give_<Attribute_3>_To);
private
   type <Object>_Representation;
   -- incomplete type, defined in package body

   type <Object> is access <Object>_Representation;
   -- pointer to an <object> representation

end <Object>_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

pragma Page;
```

```
--|*****************************************************************
--| Module Name:
--|    <object> Object Manager
--|
--| Module Type:
--|    Package Body
--|
--|-----------------------------------------------------------------
--| Module Description:
--| The Engine <object> object manager provides a means to create
--| an <object> object via the New_<object> entry and returns
--| an identification for the <object>, which is to be used when
--| updating/accessing the <object> objects state as described below.
--|
--| The Engine <object> object manager provides a means to update the
--| state of the object via the:
--|        1) Give_<input>_<state_1>_To
--|        2) Give_<input>_<state_2>_To
--|        3) Give_<input>_<state_3>_To
--|      4) Give_<attribute_1>_To
--|        5) Give_<attribute_2>_To
--|        6) Give_<attribute_3>_To
--| entries, requiring the following external state information:
--|      1) <input>_<type_1>       <type_1_units>
--|         <input>_<type_2>       <type_2_units>
--|           <input>_<type_3>     <type_3_units>
--|      2) <input>_<type_4>       <type_4_units>
--|         <input>_<type_5>       <type_5_units>
--|           <input>_<type_6>     <type_6_units>
--|      3) <input>_<type_7>       <type_7_units>
--|         <input>_<type_8>       <type_8_units>
--|           <input>_<type_9>     <type_9_units>
--|        4) <attribute_1>        <attribute_1_units>
--|        5) <attribute_2>        <attribute_2_units>
--|        6) <attribute_3>        <attribute_3_units>
--|
--| The Engine <object> object manager provides a means of obtaining
--| state information via the:
--|        1) Get_<output>_<state_1>_From
--|        2) Get_<output>_<state_2>_From
--|        3) Get_<output>_<state_3>_From
--| entries, yielding the following internal state information:
--|        1) <output>_<type_1>    <type_1_units>
--|           <output>_<type_2>    <type_2_units>
--|           <output>_<type_3>    <type_3_units>
--|        2) <output>_<type_4>    <type_4_units>
--|           <output>_<type_5>    <type_5_units>
--|           <output>_<type_6>    <type_6_units>
--|        3) <output>_<type_7>    <type_7_units>
--|           <output>_<type_8>    <type_8_units>
--|           <output>_<type_9>    <type_9_units>
--|
--| References:
--|    Design Documents:
--|        none
--|
--|    Testing and Validation:
--|        none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--| 24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
```

```
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|*****************************************************************

package body <Object>_Object_Manager is

  type <Object>_Representation is
    record
      <Input>_<Type_1> : Set.<Type_1> := ??;
      <Input>_<Type_2> : Set.<Type_2> := ??;
      <Input>_<Type_3> : Set.<Type_3> := ??;
      <Input>_<Type_4> : Set.<Type_4> := ??;
      <Input>_<Type_5> : Set.<Type_5> := ??;
      <Input>_<Type_6> : Set.<Type_6> := ??;
      <Input>_<Type_7> : Set.<Type_7> := ??;
      <Input>_<Type_8> : Set.<Type_8> := ??;
      <Input>_<Type_9> : Set.<Type_9> := ??;
      The_<Attribute_1> : <Attribute_1> := ??;
      The_<Attribute_2> : <Attribute_2> := ??;
      The_<Attribute_3> : <Attribute_3> := ??;
      <Output>_<Type_1> : Set.<Type_1> := ??;
      <Output>_<Type_2> : Set.<Type_2> := ??;
      <Output>_<Type_3> : Set.<Type_3> := ??;
      <Output>_<Type_4> : Set.<Type_4> := ??;
      <Output>_<Type_5> : Set.<Type_5> := ??;
      <Output>_<Type_6> : Set.<Type_6> := ??;
      <Output>_<Type_7> : Set.<Type_7> := ??;
      <Output>_<Type_8> : Set.<Type_8> := ??;
      <Output>_<Type_9> : Set.<Type_9> := ??;
    end record ;


  function New_<Object> return <Object> is
    --|***************************************************************
    --| Description:
    --|    This function returns a pointer to a new <object> object
    --|    representation. This pointer will be used to identify
    --|    the object for state update and state reporting purposes.
    --|
    --| Parameter Description:
    --|    return <object> which is an access to a <object> object.
    --|
    --| Notes:
    --|  none
    --|***************************************************************

  begin
    --
    -- function body goes here
    --
    RETURN null;
  end New_<Object>;



  procedure Give_<Input>_<State_1>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_1> : in Set.<Type_1>;
                  Given_<Input>_<Type_2> : in Set.<Type_2>;
                  Given_<Input>_<Type_3> : in Set.<Type_3>) is
    --|***************************************************************
```

```
  --| Description:
  --|    Initiates a change in the specified <object> object's
  --|    state given the <input>_<type_1>, <input>_<type_2>,
  --|    and the <input>_<type_3>.
  --|
  --| Parameter Description:
  --|    A_<object> identifies the <object> whose state is to be changed.
  --|    Given_<input>_<type_1> is the <input> <type_1>, in <type_1_units>
  --|    Given_<input>_<type_2> is the <input> <type_2>, in <type_2_units>
  --|    Given_<input>_<type_3> is the <input> <type_3>, in <type_3_units>
  --|
  --| Notes:
  --|   none
  --|*****************************************************************

begin
  null;
  --
  -- procedure body goes here
  --
end Give_<Input>_<State_1>_To;



procedure Give_<Input>_<State_2>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_4> : in Set.<Type_4>;
                  Given_<Input>_<Type_5> : in Set.<Type_5>;
                  Given_<Input>_<Type_6> : in Set.<Type_6>) is
  --|*****************************************************************
  --| Description:
  --|    Initiates a change in the specified <object> object's
  --|    state given the <input>_<type_4>, <input>_<type_5>,
  --|    and the <input>_<type_6>.
  --|
  --| Parameter Description:
  --|    A_<object> identifies the <object> whose state is to be changed.
  --|    Given_<input>_<type_4> is the <input> <type_4>, in <type_4_units>
  --|    Given_<input>_<type_5> is the <input> <type_5>, in <type_5_units>
  --|    Given_<input>_<type_6> is the <input> <type_6>, in <type_6_units>
  --|
  --| Notes:
  --|   none
  --|*****************************************************************

begin
  null;
  --
  -- procedure body goes here
  --
end Give_<Input>_<State_2>_To;


procedure Give_<Input>_<State_3>_To (A_<Object> : in <Object>;
                  Given_<Input>_<Type_7> : in Set.<Type_7>;
                  Given_<Input>_<Type_8> : in Set.<Type_8>;
                  Given_<Input>_<Type_9> : in Set.<Type_9>) is
  --|*****************************************************************
  --| Description:
  --|    Initiates a change in the specified <object> object's
  --|    state given the <input>_<type_7>, <input>_<type_8>,
  --|    and the <input>_<type_9>.
  --|
  --| Parameter Description:
  --|    A_<object> identifies the <object> whose state is to be changed.
  --|    Given_<input>_<type_7> is the <input> <type_7>, in <type_7_units>
```

```
   --|    Given_<input>_<type_8> is the <input> <type_8>, in <type_8_units>
   --|    Given_<input>_<type_9> is the <input> <type_9>, in <type_9_units>
   --|
   --| Notes:
   --|  none
   --|*****************************************  ******************

begin
   null;
   --
   -- procedure body goes here
   --
end Give_<Input>_<State_3>_To;



procedure Get_<Output>_<State_1>_From
         (A_<Object> : in <Object>;
          Returning_<Output>_<Type_1> : out Set.<Type_1>;
          Returning_<Output>_<Type_2> : out Set.<Type_2>;
          Returning_<Output>_<Type_3> : out Set.<Type_3>) is
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified <object> object's
   --|    state returning the <output>_<type_1>,
   --|    <output>_<type_2>, and the <output>_<type_3>.
   --|
   --| Parameter Description:
   --|    A_<object> identifies the <object> whose state is needed.
   --|    Returning_<output>_<type_1> is the <output>_<type_1> portion
   --|       of <object> object's state, in <type_1_units>
   --|    Returning_<output>_<type_2> is the <output>_<type_2> portion
   --|       of <object> object's state, in <type_2_units>
   --|    Returning_<output>_<type_3> is the <output>_<type_3> portion
   --|       of <object> object's state, in <type_3_units>
   --|
   --| Notes:
   --|  none
   --|*****************************************************************

begin
   null;
   --
   -- procedure body goes here
   --
   Returning_<Output>_<Type_1> := ??;
   Returning_<Output>_<Type_2> := ??;
   Returning_<Output>_<Type_3> := ??;
end Get_<Output>_<State_1>_From;



procedure Get_<Output>_<State_2>_From
         (A_<Object> : in <Object>;
          Returning_<Output>_<Type_4> : out Set.<Type_4>;
          Returning_<Output>_<Type_5> : out Set.<Type_5>;
          Returning_<Output>_<Type_6> : out Set.<Type_6>) is
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified <object> object's
   --|    state returning the <output>_<type_4>,
   --|    <output>_<type_5>, and the <output>_<type_6>.
   --|
   --| Parameter Description:
   --|    A_<object> identifies the <object> whose state is needed.
   --|    Returning_<output>_<type_4> is the <output>_<type_4> portion
```

```
   --|       of <object> object's state, in <type_4_units>
   --|    Returning_<output>_<type_5> is the <output>_<type_5> portion
   --|       of <object> object's state, in <type_5_units>
   --|    Returning_<output>_<type_6> is the <output>_<type_6> portion
   --|       of <object> object's state, in <type_6_units>
   --|
   --| Notes:
   --|   none
   --|*****************************************************************

begin
   null ;
   --
   -- procedure body goes here
   --
   Returning_<Output>_<Type_4> := ??;
   Returning_<Output>_<Type_5> := ??;
   Returning_<Output>_<Type_6> := ??;
end Get_<Output>_<State_2>_From;



procedure Get_<Output>_<State_3>_From
         (A_<Object> : in <Object>;
          Returning_<Output>_<Type_7> : out Set.<Type_7>;
          Returning_<Output>_<Type_8> : out Set.<Type_8>;
          Returning_<Output>_<Type_9> : out Set.<Type_9>) is
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified <object> object's
   --|    state returning the <output>_<type_7>,
   --|    <output>_<type_8>, and the <output>_<type_9>.
   --|
   --| Parameter Description:
   --|    A_<object> identifies the <object> whose state is needed.
   --|    Returning_<output>_<type_7> is the <output>_<type_7> portion
   --|       of <object> object's state, in <type_7_units>
   --|    Returning_<output>_<type_8> is the <output>_<type_8> portion
   --|       of <object> object's state, in <type_8_units>
   --|    Returning_<output>_<type_9> is the <output>_<type_9> portion
   --|       of <object> object's state, in <type_9_units>
   --|
   --| Notes:
   --|   none
   --|*****************************************************************

begin
   null ;
   --
   -- procedure body goes here
   --
   Returning_<Output>_<Type_7> := ??;
   Returning_<Output>_<Type_8> := ??;
   Returning_<Output>_<Type_9> := ??;
end Get_<Output>_<State_3>_From;



procedure Give_<Attribute_1>_To (A_<Object> : in <Object>;
                  Given_<Attribute_1> : in <Attribute_1>) is
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified <object> object's
   --|    state given the <attribute_1>.
   --|
   --| Parameter Description:
```

```
    --|    A_<object> identifies the <object> whose state is to be changed.
    --|    Given_<attribute_1> is the <attribute_1>, in <attribute_1_units>
    --|
    --| Notes:
    --|   none
    --|*****************************************************************

  begin
    null;
    --
    -- procedure body goes here
    --
  end Give_<Attribute_1>_To;


  procedure Give_<Attribute_2>_To (
        A_<Object> : in <Object>; Given_<Attribute_2> : in <Attribute_2>) is
    --|*****************************************************************
    --| Description:
    --|    Initiates a change in the specified <object> object's
    --|    state given the <attribute_2>.
    --|
    --| Parameter Description:
    --|    A_<object> identifies the <object> whose state is to be changed.
    --|    Given_<attribute_2> is the <attribute_2>, in <attribute_2_units>
    --|
    --| Notes:
    --|   none
    --|*****************************************************************

  begin
    null;
    --
    -- procedure body goes here
    --
  end Give_<Attribute_2>_To


  procedure Give_<Attribute_3>_To (
        A_<Object> : in <Object>; Given_<Attribute_3> : in <Attribute_3>) is
    --|*****************************************************************
    --| Description:
    --|    Initiates a change in the specified <object> object's
    --|    state given the <attribute_3>.
    --|
    --| Parameter Description:
    --|    A_<object> identifies the <object> whose state is to be changed.
    --|    Given_<attribute_3> is the <attribute_3>, in <attribute_3_units>
    --|
    --| Notes:
    --|   none
    --|*****************************************************************

  begin
    null;
    --
    -- procedure body goes here
    --
  end Give_<Attribute_3>_To;

end <Object>_Object_Manager;
```

Appendix C: Engine code

The Ada code that follows implements a simulator Engine system. The implementation is complete through the package specifications. The intent is to demonstrate the software architecture defined by the paradigm discussed in Chapter 4.

C.1. Package Global_Types

```
--|*****************************************************************
--| Module Name:
--|    Global Types
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    provide global types for use throughout the simulator code
--|-----------------------------------------------------------------
--| Module Description:
--|    This package provides global types for use throughout the simulator
--|    code.  The types include those necessary for compliance with the
--|    Boeing ASVP Ada code.
--|
--|    Type Execution_Sequence defines the frames to be used by the
--|    executives during the cyclic execution of the code.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    24Apr87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
```

```
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

package Global_Types is

   type Execution_Sequence is (Frame_1_Modules_Are_Executed,
                               Frame_2_Modules_Are_Executed,
                               Frame_3_Modules_Are_Executed,
                               Frame_4_Modules_Are_Executed,
                               Frame_5_Modules_Are_Executed,
                               Frame_6_Modules_Are_Executed,
                               Frame_7_Modules_Are_Executed,
                               Frame_8_Modules_Are_Executed);

end Global_Types;
```

C.2. Package Standard_Engineering_Types

```
--|*****************************************************************
--| Module Name:
--|    Standard_Engineering_Types
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package defines some standard engineering symbols and units
--| which are used in the Flight_Executive.
--|-----------------------------------------------------------------
--| Module Description:
--|    The standard engineering symbols, thier range and units of measure
--| are specified in this package. All objects and types in the
--| flight system which are represented in the real world in these units
--| should be derived from these types. New derived types can be expressed
--| as follows:
--|    type My_Blark is new Standard_Engineering_Types.Blark;
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    25AUG87    cpp    creation
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
```

```
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

package Standard_Engineering_Types is

   type Pressure is digits 6 range 0.0 .. 10000.0;
   -- pound per square inch
   type Temperature is range 300 .. 3000;
   -- degrees Rankine
   type Air_Flow is digits 4 range 0.0 .. 500.0;
   -- pounds per second
   type Thrust is digits 6 range 0.0 .. 20250.0;
   -- pounds
   type Rpm is range 0 .. 20000;
   -- revolutions per minute
   type Torque is range 0 .. 10000;
   -- pound feet

end Standard_Engineering_Types;
```

C.3. Package Bleed_Valve_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Bleed_Valve_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Bleed_Valve for the C-141 simulator.
--|    This management entails creation of Bleed_Valve object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|    The  Bleed_Valve object manager provides a means to create
--|    an Bleed_Valve object via the New_Bleed_Valve operation and returns
--|    an identification for the Bleed_Valve, which is to be used when
--|    updating/accessing the Bleed_Valve object's state as described below.
--|
--|    The Bleed_Valve object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Inlet_Air_Flow_To
--|       2) Give_Inlet_Pressure_To
--|    operations, requiring the following external state information:
--|    1) Inlet_Air_Flow  pounds per second
--|    2) Inlet_Pressure  pounds per square inch
--|
--|    The Bleed_Valve object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Discharge_Air_Flow_From
--|       2) Get_Discharge_Pressure_From
--|    operations, yielding the following internal state information:
--|       1) Discharge_Air_Flow  pounds per second
--|       2) Discharge_Pressure  pounds per square inch
--|
--| References:
--|    none
--|
--| Design Documents:
```

```
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
--|    none
--|
--| Notes:
--|    none
--|
--|------------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*********************************************************************
--|
with Standard_Engineering_Types;

package Bleed_Valve_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type Bleed_Valve is private ;
   -- an Bleed_Valve is an abstraction of a Bleed_Valve within an Engine.

   function New_Bleed_Valve return Bleed_Valve;
   --|*****************************************************************
   --| Description:
   --|    This function returns a pointer to a new Bleed_Valve object
   --|    representation. This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return Bleed_Valve which is an access to a Bleed_Valve object.
   --|*****************************************************************


   procedure Give_Inlet_Air_Flow_To (A_Bleed_Valve : in Bleed_Valve;
                                     Given_Inlet_Air_Flow : in Set.Air_Flow);
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified Bleed_Valve object's
   --|    state given the Inlet_Air_Flow.
   --|
   --| Parameter Description:
   --|    A_Bleed_Valve identifies the Bleed_Valve whose state is
   --|    to be changed.
   --|    Given_Inlet_Air_Flow is the Inlet Air_Flow, in pounds per second
   --|*****************************************************************

   procedure Give_Inlet_Pressure_To (A_Bleed_Valve : in Bleed_Valve;
                                     Given_Inlet_Pressure : in Set.Pressure);
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified Bleed_Valve object's
```

```
   --|    state given the Inlet_Pressure.
   --|
   --| Parameter Description:
   --|    A_Bleed_Valve identifies the Bleed_Valve whose state is
   --|    to be changed.
   --|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
   --|    square inch
   --|*********************************************************************

   procedure Get_Discharge_Air_Flow_From
           (A_Bleed_Valve : in Bleed_Valve;
            Returning_Discharge_Air_Flow : out Set.Air_Flow);
   --|*********************************************************************
   --| Description:
   --|    Initiates a report of the specified Bleed_Valve object's
   --|    state returning the Discharge_Air_Flow.
   --|
   --| Parameter Description:
   --|    A_Bleed_Valve identifies the Bleed_Valve whose state is needed.
   --|    Returning_Discharge_Air_Flow is the Discharge_Air_Flow portion
   --|       of Bleed_Valve object's state, in pounds per second
   --|*********************************************************************

   procedure Get_Discharge_Pressure_From
           (A_Bleed_Valve : in Bleed_Valve;
            Returning_Discharge_Pressure : out Set.Pressure);
   --|*********************************************************************
   --| Description:
   --|    Initiates a report of the specified Bleed_Valve object's
   --|    state returning the Discharge_Pressure.
   --|
   --| Parameter Description:
   --|    A_Bleed_Valve identifies the Bleed_Valve whose state is needed.
   --|    Returning_Discharge_Pressure is the Discharge_Pressure portion
   --|       of Bleed_Valve object's state, in pounds per square inch
   --|*********************************************************************


   pragma Inline (Give_Inlet_Air_Flow_To, Get_Discharge_Air_Flow_From,
            Give_Inlet_Pressure_To, Get_Discharge_Pressure_From);

private
   type Bleed_Valve_Representation;
   -- incomplete type, defined in package body

   type Bleed_Valve is access Bleed_Valve_Representation;
   -- pointer to an Bleed_Valve representation

end Bleed_Valve_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.4. Package Burner_Object_Manager

```
--|*********************************************************************
--| Module Name:
--|    Burner_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
```

```
--|    Engine Burner for the C-141 simulator.
--|    This management entails creation of Engine Burner objects,
--|    update and maintenance of its state, and finally state
--|    reporting capabilities.
--|-------------------------------------------------------------------
--| Module Description:
--|    The Engine Burner object manager provides a means to create
--|    an Burner object via the New_Burner entry and returns
--|    an identification for the Burner, which is to be used when
--|    updating/accessing the Burner objects state as described below.
--|
--|    The Engine Burner object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Inlet_Air_To
--|       2) Give_Fuel_Flow_To
--|       3) Give_Spark_To
--|    entries, requiring the following external state information:
--|       1) Inlet_Pressure     pounds per square inch
--|          Inlet_Temperature  degrees Rankine
--|          Inlet_Air_Flow     pounds per second
--|       2) The_Fuel_Flow      pounds per second
--|       3) The_Spark          joules
--|
--|    The Engine Burner object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Discharge_Air_From
--|    entries, yielding the following internal state information:
--|       1) Discharge_Pressure  pounds per square inch
--|          Discharge_Temperature degrees Rankine
--|          Discharge_Air_Flow  pounds per second
--|
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
--|    none
--|
--| Notes:
--|    none
--|
--|-------------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|-------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*******************************************************************************
--|
with Standard_Engineering_Types;

package Burner_Object_Manager is
```

```
package Set renames Standard_Engineering_Types;

type Burner is private ;
-- an Burner is an abstraction of a Burner within an Engine.

type Spark is (None, Low, High);
-- burner needs only to know relative spark size

type Fuel_Flow is (None, Flowing);
-- the burner needs to know only if it has fuel available

function New_Burner return Burner;
--|*****************************************************************
--| Description:
--|    This function returns a pointer to a new Burner object
--|    representation. This pointer will be used to identify
--|    the object for state update and state reporting purposes.
--|
--| Parameter Description:
--|    return Burner which is an access to a Burner object.
--|*****************************************************************


procedure Give_Inlet_Air_To (A_Burner : in Burner;
                  Given_Inlet_Pressure : in Set.Pressure;
                  Given_Inlet_Temperature : in Set.Temperature;
                  Given_Inlet_Air_Flow : in Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Burner object's
--|    state given the Inlet_Pressure, Inlet_Temperature,
--|    and the Inlet_Air_Flow.
--|
--| Parameter Description:
--|    A_Burner identifies the Burner whose state is to be changed.
--|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
--|       square inch
--|    Given_Inlet_Temperature is the Inlet Temperature, in degrees Rankine
--|    Given_Inlet_Air_Flow is the inlet air flow, in pounds per second,
--|*****************************************************************


procedure Get_Discharge_Air_From
        (A_Burner : in Burner;
         Returning_Discharge_Pressure : out Set.Pressure;
         Returning_Discharge_Temperature : out Set.Temperature;
         Returning_Discharge_Air_Flow : out Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Burner object's
--|    state returning the Discharge_Pressure,
--|    Discharge_Temperature, and the Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Burner identifies the Burner whose state is needed.
--|    Returning_Discharge_Pressure is the Discharge_Pressure portion
--|       of Burner object's state, in pounds per square inch
--|    Returning_Discharge_Temperature is the Discharge_Temperature portion
--|       of Burner object's state, in degrees Rankine
--|    Returning_Discharge_Air_Flow is the Discharge_Air_Flow portion
--|       of Burner object's state, in pounds per second
--|*****************************************************************


procedure Give_Fuel_Flow_To (A_Burner : in Burner;
                  Given_Fuel_Flow : in Fuel_Flow);
```

```
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified Burner object's
   --|    state given the Fuel_Flow.
   --|
   --| Parameter Description:
   --|    A_Burner identifies the Burner whose state is to be changed.
   --|    Given_Fuel_Flow is the Fuel_Flow, in pounds per second,
   --|*****************************************************************

   procedure Give_Spark_To (A_Burner : in Burner; Given_Spark : in Spark);
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified Burner object's
   --|    state given the Spark.
   --|
   --| Parameter Description:
   --|    A_Burner identifies the Burner whose state is to be changed.
   --|    Given_Spark is the Spark, in joules,
   --|*****************************************************************

   pragma Inline (Give_Inlet_Air_To, Get_Discharge_Air_From,
            Give_Fuel_Flow_To, Give_Spark_To);

private
   type Burner_Representation;
   -- incomplete type, defined in package body

   type Burner is access Burner_Representation;
   -- pointer to an Burner representation

end Burner_Object_Manager;
```

C.5. Package body Burner_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Burner Object Manager
--|
--| Module Type:
--|    Package Body
--|
--|---------------------------------------------------------------
--| Module Description:
--| The Engine Burner object manager provides a means to create
--| an Burner object via the New_Burner entry and returns
--| an identification for the Burner, which is to be used when
--| updating/accessing the Burner objects state as described below.
--|
--| The Engine Burner object manager provides a means to update the
--| state of the object via the:
--|    1) Give_Inlet_Air_To
--|    2) Give_Fuel_Flow_To
--|    3) Give_Spark_To
--| entries, requiring the following external state information:
--|    1) Inlet_Pressure     pounds per square inch
--|       Inlet_Temperature  degrees Rankine
--|       Inlet_Air_Flow     pounds per second
--|    2) The_Fuel_Flow      pounds per second
--|    3) The_Spark          joules
--|
```

```
--| The Engine Burner object manager provides a means of obtaining
--| state information via the:
--|    1) Get_Discharge_Air_From
--| entries, yielding the following internal state information:
--|    1) Discharge_Pressure  pounds per square inch
--|       Discharge_Temperature degrees Rankine
--|       Discharge_Air_Flow  pounds per second
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--| 24Aug87    cpp    Creation
--| 13Jul88 kl  Modified
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|*****************************************************************

package body Burner_Object_Manager is

   type Burner_Representation is
      record
         Inlet_Pressure : Set.Pressure := 0.0;
         Inlet_Temperature : Set.Temperature := 300;
         Inlet_Air_Flow : Set.Air_Flow := 0.0;
         The_Spark : Spark := High;
         The_Fuel_Flow : Fuel_Flow := Flowing;
         Discharge_Pressure : Set.Pressure := 0.0;
         Discharge_Temperature : Set.Temperature := 300;
         Discharge_Air_Flow : Set.Air_Flow := 0.0;
      end record ;

   function New_Burner return Burner is
      --|*****************************************************************
      --| Description:
      --|    This function returns a pointer to a new Burner object
      --|    representation.  This pointer will be used to identify
      --|    the object for state update and state reporting purposes.
      --|
      --| Parameter Description:
      --|    return Burner which is an access to a Burner object.
      --|
      --| Notes:
      --|
      --|*****************************************************************

   begin
      --
      -- function body goes here
      --
      RETURN null ;
   end New_Burner;
```

```
procedure Give_Inlet_Air_To (A_Burner : in Burner;
                             Given_Inlet_Pressure : in Set.Pressure;
                             Given_Inlet_Temperature : in Set.Temperature;
                             Given_Inlet_Air_Flow : in Set.Air_Flow) is
  --|*************************************************************
  --| Description:
  --|    Initiates a change in the specified Burner object's
  --|    state given the Inlet_Pressure, Inlet_Temperature,
  --|    and the Inlet_Air_Flow.
  --|
  --| Parameter Description:
  --|    A_Burner identifies the Burner whose state is to be changed.
  --|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
  --|       square inch
  --|    Given_Inlet_Temperature is the Inlet Temperature, in degrees Rankine
  --|    Given_Inlet_Air_Flow is the inlet air flow, in pounds per second,
  --|
  --| Notes:
  --|
  --|*************************************************************

begin
  null;
  --
  -- procedure body goes here
  --
end Give_Inlet_Air_To;


procedure Get_Discharge_Air_From
        (A_Burner : in Burner;
         Returning_Discharge_Pressure : out Set.Pressure;
         Returning_Discharge_Temperature : out Set.Temperature;
         Returning_Discharge_Air_Flow : out Set.Air_Flow) is
  --|*************************************************************
  --| Description:
  --|    Initiates a report of the specified Burner object's
  --|    state returning the Discharge_Pressure,
  --|    Discharge_Temperature, and the Discharge_Air_Flow.
  --|
  --| Parameter Description:
  --|    A_Burner identifies the Burner whose state is needed.
  --|    Returning_Discharge_Pressure is the Discharge_Pressure portion
  --|       of Burner object's state, in pounds per square inch
  --|    Returning_Discharge_Temperature is the Discharge_Temperature portion
  --|       of Burner object's state, in degrees Rankine
  --|    Returning_Discharge_Air_Flow is the Discharge_Air_Flow portion
  --|       of Burner object's state, in pounds per second
  --|
  --| Notes:
  --|
  --|*************************************************************

begin
  null;
  --
  -- procedure body goes here
  --
  Returning_Discharge_Pressure := 0.0;
  Returning_Discharge_Temperature := 300;
  Returning_Discharge_Air_Flow := 0.0;
end Get_Discharge_Air_From;


procedure Give_Fuel_Flow_To (A_Burner : in Burner;
```

```
                    Given_Fuel_Flow : in  Fuel_Flow) is
    --|*****************************************************************
    --| Description:
    --|    Initiates a change in the specified Burner object's
    --|    state given the Fuel_Flow.
    --|
    --| Parameter Description:
    --|    A_Burner identifies the Burner whose state is to be changed.
    --|    Given_Fuel_Flow is the Fuel_Flow, in pounds per second,
    --|
    --| Notes:
    --|
    --|*****************************************************************

  begin
    null;
    --
    -- procedure body goes here
    --
  end Give_Fuel_Flow_To;


  procedure Give_Spark_To (A_Burner : in Burner; Given_Spark : in Spark) is
    --|*****************************************************************
    --| Description:
    --|    Initiates a change in the specified Burner object's
    --|    state given the Spark.
    --|
    --| Parameter Description:
    --|    A_Burner identifies the Burner whose state is to be changed.
    --|    Given_Spark is the Spark, in joules,
    --|
    --| Notes:
    --|
    --|*****************************************************************

  begin
    null;
    --
    -- procedure body goes here
    --
  end Give_Spark_To;

end Burner_Object_Manager;
```

C.6. Package Diffuser_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Diffuser_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Diffuser for the C-141 simulator.
--|    This management entails creation of Diffuser object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
```

```
--|   The  Diffuser object manager provides a means to create
--|   an Diffuser object via the New_Diffuser operation and returns
--|   an identification for the Diffuser, which is to be used when
--|   updating/accessing the Diffuser object's state as described below.
--|
--|   The Diffuser object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Pressure_and_Temperature_To
--|   2) Give_Mach_Number_To
--|   operations, requiring the following external state information:
--|   1) Inlet_Pressure  pounds per square inch
--|   Inlet_Temperature  degrees Rankine
--|      2) The_Mach_Number    <dimensionless>
--|
--|   The Diffuser object manager provides a means of obtaining
--|   state information via the:
--|      1) Get_Discharge_Air_From
--|   operations, yielding the following internal state information:
--|      1) Discharge_Pressure  pounds per square inch
--|         Discharge_Temperature  degrees Rankine
--|         Discharge_Air_Flow  pounds per second
--|
--| References:
--|   none
--|
--| Design Documents:
--|   none
--|
--| User's Manual:
--|   none
--|
--| Testing and Validation:
--|   none
--|
--| Notes:
--|   none
--|
--|------------------------------------------------------------------------
--| Modification History:
--|   24Aug87   cpp   Creation
--| 13Jul88   kl    Modified
--|
--|------------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   Distribution unlimited
--|
--|   No warranty is implied.
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense."
--|
--|************************************************************************

with Standard_Engineering_Types;

package Diffuser_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type Diffuser is private ;
   -- an Diffuser is an abstraction of a Diffuser within an Engine.

   type Mach_Number is digits 3 range 0.00 .. 1.00;
   --<dimensionless>

   function New_Diffuser return Diffuser;
```

```
--|*****************************************************************
--| Description:
--|    This function returns a pointer to a new Diffuser object
--|    representation.  This pointer will be used to identify
--|    the object for state update and state reporting purposes.
--|
--| Parameter Description:
--|    return Diffuser which is an access to a Diffuser object.
--|*****************************************************************


procedure Give_Inlet_Pressure_And_Temperature_To
        (A_Diffuser : in Diffuser;
         Given_Inlet_Pressure : in Set.Pressure;
         Given_Inlet_Temperature : in Set.Temperature);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Diffuser object's
--|    state given the Inlet_Pressure, Inlet_Temperature.
--|
--| Parameter Description:
--|    A_Diffuser identifies the Diffuser whose state is to be changed.
--|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
--|    square inch
--|    Given_Inlet_Temperature is the Inlet Temperature, in degrees Rankine
--|*****************************************************************


procedure Get_Discharge_Air_From
        (A_Diffuser : in Diffuser;
         Returning_Discharge_Pressure : out Set.Pressure;
         Returning_Discharge_Temperature : out Set.Temperature;
         Returning_Discharge_Air_Flow : out Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Diffuser object's
--|    state returning the Discharge_Pressure,
--|    Discharge_Temperature, and the Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Diffuser identifies the Diffuser whose state is needed.
--|    Returning_Discharge_Pressure is the Discharge_Pressure portion
--|       of Diffuser object's state, in pounds per square inch
--|    Returning_Discharge_Temperature is the Discharge_Temperature portion
--|       of Diffuser object's state, in degrees Rankine
--|    Returning_Discharge_Air_Flow is the Discharge_Air_Flow portion
--|       of Diffuser object's state, in pounds per second
--|*****************************************************************


procedure Give_Mach_Number_To (A_Diffuser : in Diffuser;
                               Given_Mach_Number : in Mach_Number);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Diffuser object's
--|    state given the Mach_Number.
--|
--| Parameter Description:
--|    A_Diffuser identifies the Diffuser whose state is to be changed.
--|    Given_Mach_Number is the Mach_Number, in <dimensionless>
--|*****************************************************************


pragma Inline (Give_Inlet_Pressure_And_Temperature_To,
        Get_Discharge_Air_From, Give_Mach_Number_To);
```

```
private
   type Diffuser_Representation;
   -- incomplete type, defined in package body

   type Diffuser is access Diffuser_Representation;
   -- pointer to an Diffuser representation

end Diffuser_Object_Manager;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.7. Package Engine_Casing_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Engine_Casing_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Engine_Casing for the C-141 simulator.
--|    This management entails creation of Engine_Casing object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|    The  Engine_Casing object manager provides a means to create
--|    an Engine_Casing object via the New_Engine_Casing operation and returns
--|    an identification for the Engine_Casing, which is to be used when
--|    updating/accessing the Engine_Casing object's state as described below.
--|
--|    The Engine_Casing object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Inlet_Pressure_To
--|       2) Give_Inlet_Air_Flow_To
--|       3) Give_Inlet_Temperature_To
--|    operations, requiring the following external state information:
--|    1) Inlet_Pressure     pounds per square inch
--|    2) Inlet_Air_Flow     pounds per second
--|    3) Inlet_Temperature  degress Rankine
--|
--|    The Engine_Casing object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Discharge_Pressure_From
--|       2) Get_Discharge_Air_Flow_From
--|       3) Get_Discharge_Temperature_From
--|    operations, yielding the following internal state information:
--|       1) Discharge_Pressure  pounds per square inch
--|       2) Discharge_Air_Flow  pounds per second
--|       3) Discharge_Temperature  degress Rankine
--|
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
```

```
--|    none
--|
--| Notes:
--|    none
--|
--|-------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*************************************************************************
--|
with Standard_Engineering_Types;

package Engine_Casing_Object_Manager is

  package Set renames Standard_Engineering_Types;

  type Engine_Casing is private ;
  -- an Engine_Casing is an abstraction of a Engine_Casing within an Engine.

  function New_Engine_Casing return Engine_Casing;
  --|*********************************************************************
  --| Description:
  --|    This function returns a pointer to a new Engine_Casing object
  --|    representation.  This pointer will be used to identify
  --|    the object for state update and state reporting purposes.
  --|
  --| Parameter Description:
  --|    return Engine_Casing which is an access to a Engine_Casing object.
  --|*********************************************************************


  procedure Give_Inlet_Pressure_To (A_Engine_Casing : in Engine_Casing;
                                   Given_Inlet_Pressure : in Set.Pressure);
  --|*********************************************************************
  --| Description:
  --|    Initiates a change in the specified Engine_Casing object's
  --|    state given the Inlet_Pressure.
  --|
  --| Parameter Description:
  --|    A_Engine_Casing identifies the Engine_Casing whose state is
  --|    to be changed.
  --|    Given_Inlet_Pressure is the Inlet Pressure, in pounds
  --|    per square inch
  --|*********************************************************************

  procedure Give_Inlet_Air_Flow_To (A_Engine_Casing : in Engine_Casing;
                                   Given_Inlet_Air_Flow : in Set.Air_Flow);
  --|*********************************************************************
  --| Description:
  --|    Initiates a change in the specified Engine_Casing object's
  --|    state given the Inlet_Air_Flow.
  --|
  --| Parameter Description:
  --|    A_Engine_Casing identifies the Engine_Casing whose state is
  --|     to be changed.
```

```
--|    Given_Inlet_Air_Flow is the Inlet Air_Flow, in pounds per second
--|*****************************************************************

procedure Give_Inlet_Temperature_To
        (A_Engine_Casing : in Engine_Casing;
         Given_Inlet_Temperature : in Set.Temperature);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Engine_Casing object's
--|    state given the Inlet_Temperature.
--|
--| Parameter Description:
--|    A_Engine_Casing identifies the Engine_Casing whose state is
--|    to be changed.
--|    Given_Inlet_Temperature is the Inlet Temperature, in degress Rankine
--|*****************************************************************


procedure Get_Discharge_Pressure_From
        (A_Engine_Casing : in Engine_Casing;
         Returning_Discharge_Pressure : out Set.Pressure);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Engine_Casing object's
--|    state returning the Discharge_Pressure.
--|
--| Parameter Description:
--|    A_Engine_Casing identifies the Engine_Casing whose state is needed.
--|    Returning_Discharge_Pressure is the Discharge_Pressure portion
--|       of Engine_Casing object's state, in pounds per square inch
--|*****************************************************************

procedure Get_Discharge_Air_Flow_From
        (A_Engine_Casing : in Engine_Casing;
         Returning_Discharge_Air_Flow : out Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Engine_Casing object's
--|    state returning the Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Engine_Casing identifies the Engine_Casing whose state is needed.
--|    Returning_Discharge_Air_Flow is the Discharge_Air_Flow portion
--|       of Engine_Casing object's state, in pounds per second
--|*****************************************************************

procedure Get_Discharge_Temperature_From
        (A_Engine_Casing : in Engine_Casing;
         Returning_Discharge_Temperature : out Set.Temperature);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Engine_Casing object's
--|    state returning the Discharge_Temperature.
--|
--| Parameter Description:
--|    A_Engine_Casing identifies the Engine_Casing whose state is needed.
--|    Returning_Discharge_Temperature is the Discharge_Temperature portion
--|       of Engine_Casing object's state, in degress Rankine
--|*****************************************************************


pragma Inline (Give_Inlet_Pressure_To, Get_Discharge_Pressure_From,
          Give_Inlet_Air_Flow_To, Get_Discharge_Air_Flow_From,
          Give_Inlet_Temperature_To, Get_Discharge_Temperature_From);
```

```
private
   type Engine_Casing_Representation;
   -- incomplete type, defined in package body

   type Engine_Casing is access Engine_Casing_Representation;
   -- pointer to an Engine_Casing representation

end Engine_Casing_Object_Manager;

--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.8. Package Exhaust_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Exhaust_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Exhaust for the C-141 simulator.
--|    This management entails creation of Exhaust object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|    The  Exhaust object manager provides a means to create
--|    an Exhaust object via the New_Exhaust operation and returns
--|    an identification for the Exhaust, which is to be used when
--|    updating/accessing the Exhaust object's state as described below.
--|
--|    The Exhaust object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Inlet_Pressure_To
--|    operations, requiring the following external state information:
--|    1) Inlet_Pressure  pounds per square inch
--|
--|    The Exhaust object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Discharge_Thrust_From
--|       2) Get_Egt_From
--|       3) Get_Epr_From
--|    operations, yielding the following internal state information:
--|       1) Discharge_Thrust    pounds per square inch
--|       2) Egt degrees Rankine
--|       3) The_Epr <diemensionless>
--|
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
--|    none
--|
--| Notes:
--|    none
```

```
--|
--|-------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*************************************************************************
--|
with Standard_Engineering_Types;

package Exhaust_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type Exhaust is private ;
   -- an Exhaust is an abstraction of a Exhaust within an Engine.

   type Epr is digits 2 range 1.2 .. 2.3;
   -- <dimensionless>

   function New_Exhaust return Exhaust;
   --|*********************************************************************
   --| Description:
   --|    This function returns a pointer to a new Exhaust object
   --|    representation. This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return Exhaust which is an access to a Exhaust object.
   --|*********************************************************************


   procedure Give_Inlet_Pressure_To (A_Exhaust : in Exhaust;
                                    Given_Inlet_Pressure : in Set.Pressure);
   --|*********************************************************************
   --| Description:
   --|    Initiates a change in the specified Exhaust object's
   --|    state given the Inlet_Pressure.
   --|
   --| Parameter Description:
   --|    A_Exhaust identifies the Exhaust whose state is to be changed.
   --|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
   --|    square inch
   --|*********************************************************************

   procedure Get_Discharge_Thrust_From
           (A_Exhaust : in Exhaust;
            Returning_Discharge_Thrust : out Set.Pressure);
   --|*********************************************************************
   --| Description:
   --|    Initiates a report of the specified Exhaust object's
   --|    state returning the Discharge_Thrust.
   --|
   --| Parameter Description:
   --|    A_Exhaust identifies the Exhaust whose state is needed.
   --|    Returning_Discharge_Thrust is the Discharge_Thrust portion
   --|       of Exhaust object's state, in pounds per square inch
```

```
   --|*****************************************************************

   procedure Get_Egt_From (A_Exhaust : in Exhaust;
                           Returning_Egt : out Set.Temperature);
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified Exhaust object's
   --|    state returning the Egt.
   --|
   --| Parameter Description:
   --|    A_Exhaust identifies the Exhaust whose state is needed.
   --|    Returning_Egt is the Egt portion
   --|       of Exhaust object's state, in degrees Rankine
   --|*****************************************************************

   procedure Get_Epr_From (A_Exhaust : in Exhaust; Returning_Epr : out Epr);
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified Exhaust object's
   --|    state returning the Epr.
   --|
   --| Parameter Description:
   --|    A_Exhaust identifies the Exhaust whose state is needed.
   --|    Returning_Epr is the Epr portion
   --|       of Exhaust object's state, in <diemensionless>
   --|*****************************************************************



   pragma Inline (Give_Inlet_Pressure_To, Get_Discharge_Thrust_From,
            Get_Egt_From, Get_Epr_From);

private
   type Exhaust_Representation;
   -- incomplete type, defined in package body

   type Exhaust is access Exhaust_Representation;
   -- pointer to an Exhaust representation

end Exhaust_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.9. Package Fan_Duct_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Fan_Duct_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Fan_Duct for the C-141 simulator.
--|    This management entails creation of Fan_Duct object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|    The Fan_Duct object manager provides a means to create
--|    an Fan_Duct object via the New_Fan_Duct operation and returns
--|    an identification for the Fan_Duct, which is to be used when
```

```
--|    updating/accessing the Fan_Duct object's state as described below.
--|
--|    The Fan_Duct object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Inlet_Pressure_To
--|    operations, requiring the following external state information:
--|    1) Inlet_Pressure  pounds per square inch
--|
--|    The Fan_Duct object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Discharge_Thrust_From
--|    operations, yielding the following internal state information:
--|       1) Discharge_Thrust    pounds
--|
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
--|    none
--|
--| Notes:
--|    none
--|
--|-----------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************************
--|
with Standard_Engineering_Types;

package Fan_Duct_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type Fan_Duct is private ;
   -- an Fan_Duct is an abstraction of a Fan_Duct within an Engine.

   function New_Fan_Duct return Fan_Duct;
   --|*****************************************************************
   --| Description:
   --|    This function returns a pointer to a new Fan_Duct object
   --|    representation. This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return Fan_Duct which is an access to a Fan_Duct object.
   --|*****************************************************************
```

```
   procedure Give_Inlet_Pressure_To (A_Fan_Duct : in Fan_Duct;
                                     Given_Inlet_Pressure : in Set.Pressure);
   --|*********************************************************************
   --| Description:
   --|    Initiates a change in the specified Fan_Duct object's
   --|    state given the Inlet_Pressure.
   --|
   --| Parameter Description:
   --|    A_Fan_Duct identifies the Fan_Duct whose state is to be changed.
   --|    Given_Inlet_Pressure is the Inlet Pressure, in pounds per
   --|    square inch
   --|*********************************************************************


   procedure Get_Discharge_Thrust_From
           (A_Fan_Duct : in Fan_Duct;
            Returning_Discharge_Thrust : out Set.Thrust);
   --|*********************************************************************
   --| Description:
   --|    Initiates a report of the specified Fan_Duct object's
   --|    state returning the Discharge_Thrust.
   --|
   --| Parameter Description:
   --|    A_Fan_Duct identifies the Fan_Duct whose state is needed.
   --|    Returning_Discharge_Thrust is the Discharge_Thrust portion
   --|       of Fan_Duct object's state, in pounds
   --|*********************************************************************


   pragma Inline (Give_Inlet_Pressure_To, Get_Discharge_Thrust_From);

private
   type Fan_Duct_Representation;
   -- incomplete type, defined in package body

   type Fan_Duct is access Fan_Duct_Representation;
   -- pointer to an Fan_Duct representation

end Fan_Duct_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.10. Package Rotor1_Object_Manager

```
--|*********************************************************************
--| Module Name:
--|    Rotor1_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Rotor1 for the C-141 simulator.
--|    This management entails creation of Rotor1 object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|---------------------------------------------------------------------
--| Module Description:
--|    The Rotor1 object manager provides a means to create
--|    an Rotor1 object via the New_Rotor1 operation and returns
--|    an identification for the Rotor1, which is to be used when
--|    updating/accessing the Rotor1 object's state as described below.
```

```
--|
--|   The Rotor1 object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Fan1_Inlet_Air_To
--|      2) Give_Turbine1_Inlet_Air_To
--|   operations, requiring the following external state information:
--|   1) Fan1_Inlet_Pressure pounds per square inch
--|   Fan1_Inlet_Temperature  degrees Rankine
--|        Fan1_Inlet_Air_Flow pounds per second
--|   2) Turbine1_Inlet_Pressure pounds per square inch
--|   Turbine1_Inlet_Temperature  degrees Rankine
--|        Turbine1_Inlet_Air_Flow pounds per second
--|
--|   The Rotor1 object manager provides a means of obtaining
--|   state information via the:
--|      1) Get_Fan1_Discharge_Air_From
--|      2) Get_Turbine1_Discharge_Air_From
--|      3) Get_Rpm_From
--|      4) Get_Vibration_From
--|   operations, yielding the following internal state information:
--|      1) Fan1_Discharge_Pressure pounds per square inch
--|         Fan1_Discharge_Temperature  degrees Rankine
--|         Fan1_Discharge_Air_Flow pounds per second
--|      2) Turbine1_Discharge_Pressure pounds per square inch
--|         Turbine1_Discharge_Temperature  degrees Rankine
--|         Turbine1_Discharge_Air_Flow pounds per second
--|      3) The_Rpm rpm
--|      4) The_Vibration   mils
--|
--| References:
--|   none
--|
--| Design Documents:
--|   none
--|
--| User's Manual:
--|   none
--|
--| Testing and Validation:
--|   none
--|
--| Notes:
--|   none
--|
--|------------------------------------------------------------------
--| Modification History:
--|   24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   Distribution unlimited
--|
--|   No warranty is implied.
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense."
--|
--|*********************************************************************
--|
with Standard_Engineering_Types;

package Rotor1_Object_Manager is

   package Set renames Standard_Engineering_Types;
```

```
type Rotor1 is private ;
-- an Rotor1 is an abstraction of a Rotor1 within an Engine.

type Vibration is range 0 .. 5;
-- mils

function New_Rotor1 return Rotor1;
--|*****************************************************************
--| Description:
--|    This function returns a pointer to a new Rotor1 object
--|    representation. This pointer will be used to identify
--|    the object for state update and state reporting purposes.
--|
--| Parameter Description:
--|    return Rotor1 which is an access to a Rotor1 object.
--|*****************************************************************


procedure Give_Fan1_Inlet_Air_To
        (A_Rotor1 : in Rotor1;
         Given_Fan1_Inlet_Pressure : in Set.Pressure;
         Given_Fan1_Inlet_Temperature : in Set.Temperature;
         Given_Fan1_Inlet_Air_Flow : in Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Rotor1 object's
--|    state given the Fan1_Inlet_Pressure, Fan1_Inlet_Temperature,
--|    and the Fan1_Inlet_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is to be changed.
--|    Given_Fan1_Inlet_Pressure is the Inlet Pressure, in pounds
--|    per square inch
--|    Given_Fan1_Inlet_Temperature is the Inlet Temperature,
--|    in degrees Rankine
--|    Given_Fan1_Inlet_Air_Flow is the Inlet Air_Flow, in pounds
--|    per second
--|*****************************************************************

procedure Give_Turbine1_Inlet_Air_To
        (A_Rotor1 : in Rotor1;
         Given_Turbine1_Inlet_Pressure : in Set.Pressure;
         Given_Turbine1_Inlet_Temperature : in Set.Temperature;
         Given_Turbine1_Inlet_Air_Flow : in Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a change in the specified Rotor1 object's
--|    state given the Turbine1_Inlet_Pressure, Turbine1_Inlet_Temperature,
--|    and the Turbine1_Inlet_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is to be changed.
--|    Given_Turbine1_Inlet_Pressure is the Inlet Pressure,
--|    in pounds per square inch
--|    Given_Turbine1_Inlet_Temperature is the Inlet Temperature,
--|    in degrees Rankine
--|    Given_Turbine1_Inlet_Air_Flow is the Inlet Air_Flow, in
--|    pounds per second
--|*****************************************************************


procedure Get_Fan1_Discharge_Air_From
        (A_Rotor1 : in Rotor1;
         Returning_Fan1_Discharge_Pressure : out Set.Pressure;
         Returning_Fan1_Discharge_Temperature : out Set.Temperature;
         Returning_Fan1_Discharge_Air_Flow : out Set.Air_Flow);
```

```
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Rotor1 object's
--|    state returning the Fan1_Discharge_Pressure,
--|    Fan1_Discharge_Temperature, and the Fan1_Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is needed.
--|    Returning_Fan1_Discharge_Pressure is the Fan1_Discharge_Pressure portion
--|       of Rotor1 object's state, in pounds per square inch
--|    Returning_Fan1_Discharge_Temperature is the Fan1_Discharge_Temperature portion
--|       of Rotor1 object's state, in degrees Rankine
--|    Returning_Fan1_Discharge_Air_Flow is the Fan1_Discharge_Air_Flow portion
--|       of Rotor1 object's state, in pounds per second
--|*****************************************************************

procedure Get_Turbine1_Discharge_Air_From
        (A_Rotor1 : in Rotor1;
         Returning_Turbine1_Discharge_Pressure : out Set.Pressure;
         Returning_Turbine1_Discharge_Temperature : out
           Set.Temperature;
         Returning_Turbine1_Discharge_Air_Flow : out Set.Air_Flow);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Rotor1 object's
--|    state returning the Turbine1_Discharge_Pressure,
--|    Turbine1_Discharge_Temperature, and the Turbine1_Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is needed.
--|    Returning_Turbine1_Discharge_Pressure is the Turbine1_Discharge_Pressure portion
--|       of Rotor1 object's state, in pounds per square inch
--|    Returning_Turbine1_Discharge_Temperature is the Turbine1_Discharge_Temperature portion
--|       of Rotor1 object's state, in degrees Rankine
--|    Returning_Turbine1_Discharge_Air_Flow is the Turbine1_Discharge_Air_Flow portion
--|       of Rotor1 object's state, in pounds per second
--|*****************************************************************

procedure Get_Rpm_From (A_Rotor1 : in Rotor1; Returning_Rpm : out Set.Rpm);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Rotor1 object's
--|    state returning the Rpm.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is needed.
--|    Returning_Rpm is the Rpm portion
--|       of Rotor1 object's state, in rpm
--|*****************************************************************


procedure Get_Vibration_From (A_Rotor1 : in Rotor1;
                              Returning_Vibration : out Vibration);
--|*****************************************************************
--| Description:
--|    Initiates a report of the specified Rotor1 object's
--|    state returning the Vibration.
--|
--| Parameter Description:
--|    A_Rotor1 identifies the Rotor1 whose state is needed.
--|    Returning_Vibration is the Vibration portion
--|       of Rotor1 object's state, in mils
--|*****************************************************************

pragma Inline (Give_Fan1_Inlet_Air_To, Get_Fan1_Discharge_Air_From,
         Give_Turbine1_Inlet_Air_To, Get_Turbine1_Discharge_Air_From,
```

```
            Get_Rpm_From, Get_Vibration_From);

private
   type Rotor1_Representation;
   -- incomplete type, defined in package body

   type Rotor1 is access Rotor1_Representation;
   -- pointer to an Rotor1 representation

end Rotor1_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.11. Package Rotor2_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|    Rotor2_Object_Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package manages objects which simulate the
--|    Engine Rotor2 for the C-141 simulator.
--|    This management entails creation of Rotor2 object's
--|    update, maintenance of its state, and state
--|    reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|    The  Rotor2 object manager provides a means to create
--|    an Rotor2 object via the New_Rotor2 operation and returns
--|    an identification for the Rotor2, which is to be used when
--|    updating/accessing the Rotor2 object's state as described below.
--|
--|    The Rotor2 object manager provides a means to update the
--|    state of the object via the:
--|       1) Give_Fan2_Inlet_Air_To
--|       2) Give_Turbine2_Inlet_Air_To
--|       3) Give_Torque_To
--|    operations, requiring the following external state information:
--|    1) Fan2_Inlet_Pressure pounds per square inch
--|    Fan2_Inlet_Temperature  degrees Rankine
--|          Fan2_Inlet_Air_Flow pounds per second
--|    2) Turbine2_Inlet_Pressure pounds per square inch
--|    Turbine2_Inlet_Temperature  degrees Rankine
--|          Turbine2_Inlet_Air_Flow pounds per second
--|       3) The_Torque  pound feet
--|
--|    The Rotor2 object manager provides a means of obtaining
--|    state information via the:
--|       1) Get_Fan2_Discharge_Air_From
--|       2) Get_Turbine2_Discharge_Air_From
--|       3) Get_Vibration_From
--|    operations, yielding the following internal state information:
--|       1) Fan2_Discharge_Pressure pounds per square inch
--|          Fan2_Discharge_Temperature  degrees Rankine
--|          Fan2_Discharge_Air_Flow pounds per second
--|       2) Turbine2_Discharge_Pressure pounds per square inch
--|          Turbine2_Discharge_Temperature  degrees Rankine
--|          Turbine2_Discharge_Air_Flow pounds per second
--|       3) The_Vibration  mils
--|
```

```
--| References:
--|    none
--|
--| Design Documents:
--|    none
--|
--| User's Manual:
--|    none
--|
--| Testing and Validation:
--|    none
--|
--| Notes:
--|    none
--|
--|------------------------------------------------------------------------
--| Modification History:
--|    24Aug87    cpp    Creation
--| 13Jul88    kl     Modified
--|
--|------------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|************************************************************************
--|
with Standard_Engineering_Types;

package Rotor2_Object_Manager is

   package Set renames Standard_Engineering_Types;

   type Rotor2 is private ;
   -- an Rotor2 is an abstraction of a Rotor2 within an Engine.

   type Vibration is range 0 .. 5;
   -- mils

   function New_Rotor2 return Rotor2;
   --|*********************************************************************
   --| Description:
   --|    This function returns a pointer to a new Rotor2 object
   --|    representation.  This pointer will be used to identify
   --|    the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return Rotor2 which is an access to a Rotor2 object.
   --|*********************************************************************


   procedure Give_Fan2_Inlet_Air_To
           (A_Rotor2 : in Rotor2;
            Given_Fan2_Inlet_Pressure : in Set.Pressure;
            Given_Fan2_Inlet_Temperature : in Set.Temperature;
            Given_Fan2_Inlet_Air_Flow : in Set.Air_Flow);
   --|*********************************************************************
   --| Description:
   --|    Initiates a change in the specified Rotor2 object's
   --|    state given the Fan2_Inlet_Pressure, Fan2_Inlet_Temperature,
   --|    and the Fan2_Inlet_Air_Flow.
   --|
```

```
--| Parameter Description:
--|    A_Rotor2 identifies the Rotor2 whose state is to be changed.
--|    Given_Fan2_Inlet_Pressure is the Inlet Pressure, in pounds
--|    per square inch
--|    Given_Fan2_Inlet_Temperature is the Inlet Temperature,
--|    in degrees Rankine
--|    Given_Fan2_Inlet_Air_Flow is the Inlet Air_Flow, in pounds
--|    per second
--|*********************************************************************

procedure Give_Turbine2_Inlet_Air_To
        (A_Rotor2 : in Rotor2;
         Given_Turbine2_Inlet_Pressure : in Set.Pressure;
         Given_Turbine2_Inlet_Temperature : in Set.Temperature;
         Given_Turbine2_Inlet_Air_Flow : in Set.Air_Flow);
--|*********************************************************************
--| Description:
--|    Initiates a change in the specified Rotor2 object's
--|    state given the Turbine2_Inlet_Pressure, Turbine2_Inlet_Temperature,
--|    and the Turbine2_Inlet_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor2 identifies the Rotor2 whose state is to be changed.
--|    Given_Turbine2_Inlet_Pressure is the Inlet Pressure,
--|    in pounds per square inch
--|    Given_Turbine2_Inlet_Temperature is the Inlet Temperature,
--|    in degrees Rankine
--|    Given_Turbine2_Inlet_Air_Flow is the Inlet Air_Flow, in
--|    pounds per second
--|*********************************************************************


procedure Get_Fan2_Discharge_Air_From
        (A_Rotor2 : in Rotor2;
         Returning_Fan2_Discharge_Pressure : out Set.Pressure;
         Returning_Fan2_Discharge_Temperature : out Set.Temperature;
         Returning_Fan2_Discharge_Air_Flow : out Set.Air_Flow);
--|*********************************************************************
--| Description:
--|    Initiates a report of the specified Rotor2 object's
--|    state returning the Fan2_Discharge_Pressure,
--|    Fan2_Discharge_Temperature, and the Fan2_Discharge_Air_Flow.
--|
--| Parameter Description:
--|    A_Rotor2 identifies the Rotor2 whose state is needed.
--|    Returning_Fan2_Discharge_Pressure is the Fan2_Discharge_Pressure portion
--|       of Rotor2 object's state, in pounds per square inch
--|    Returning_Fan2_Discharge_Temperature is the Fan2_Discharge_Temperature portion
--|       of Rotor2 object's state, in degrees Rankine
--|    Returning_Fan2_Discharge_Air_Flow is the Fan2_Discharge_Air_Flow portion
--|       of Rotor2 object's state, in pounds per second
--|*********************************************************************

procedure Get_Turbine2_Discharge_Air_From
        (A_Rotor2 : in Rotor2;
         Returning_Turbine2_Discharge_Pressure : out Set.Pressure;
         Returning_Turbine2_Discharge_Temperature : out
           Set.Temperature;
         Returning_Turbine2_Discharge_Air_Flow : out Set.Air_Flow);
--|*********************************************************************
--| Description:
--|    Initiates a report of the specified Rotor2 object's
--|    state returning the Turbine2_Discharge_Pressure,
--|    Turbine2_Discharge_Temperature, and the Turbine2_Discharge_Air_Flow.
--|
--| Parameter Description:
```

```
   --|    A_Rotor2 identifies the Rotor2 whose state is needed.
   --|    Returning_Turbine2_Discharge_Pressure is the Turbine2_Discharge_Pressure portion
   --|       of Rotor2 object's state, in pounds per square inch
   --|    Returning_Turbine2_Discharge_Temperature is the Turbine2_Discharge_Temperature portion
   --|       of Rotor2 object's state, in degrees Rankine
   --|    Returning_Turbine2_Discharge_Air_Flow is the Turbine2_Discharge_Air_Flow portion
   --|       of Rotor2 object's state, in pounds per second
   --|*****************************************************************

   procedure Get_Vibration_From (A_Rotor2 : in Rotor2;
                                 Returning_Vibration : out Vibration);
   --|*****************************************************************
   --| Description:
   --|    Initiates a report of the specified Rotor2 object's
   --|    state returning the Vibration.
   --|
   --| Parameter Description:
   --|    A_Rotor2 identifies the Rotor2 whose state is needed.
   --|    Returning_Vibration is the Vibration portion
   --|       of Rotor2 object's state, in mils
   --|*****************************************************************

   procedure Give_Torque_To (A_Rotor2 : in Rotor2; The_Torque : in Set.Torque);
   --|*****************************************************************
   --| Description:
   --|    Initiates a change in the specified Rotor2 object's
   --|    state given the The_Torque.
   --|
   --| Parameter Description:
   --|    A_Rotor2 identifies the Rotor2 whose state is to be changed.
   --|    The_Torque is the Torque, in pound feet
   --|*****************************************************************

   pragma Inline (Give_Fan2_Inlet_Air_To, Get_Fan2_Discharge_Air_From,
                  Give_Turbine2_Inlet_Air_To, Get_Turbine2_Discharge_Air_From,
                  Get_Vibration_From, Give_Torque_To);

private
   type Rotor2_Representation;
   -- incomplete type, defined in package body

   type Rotor2 is access Rotor2_Representation;
   -- pointer to an Rotor2 representation

end Rotor2_Object_Manager;

--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

C.12. Package Flight_Executive

```
--|*****************************************************************
--| Module Name:
--|    Flight Executive
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Executive for flight systems
--|-----------------------------------------------------------------
```

```
--| Module Description:
--|    This executive is responsible for processing all flight systems.
--|    Processing involves handling all connections between the flight
--|    systems and processing each system.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

with Global_Types;

package Flight_Executive is

   procedure Update_Flight_Executive
          (Frame : in Global_Types.Execution_Sequence);
--|*****************************************************************
--| Description:
--|    executive which updates all flight systems
--|
--| Parameter Description:
--|    frame is the current executing frame
--|*****************************************************************

end Flight_Executive;
```

C.13. Package body Flight_Executive

```
--|*****************************************************************
--| Module Name:
--|    Flight Executive
--|
--| Module Type:
--|    Package Body
--|
--|-----------------------------------------------------------------
--| Module Description:
--|    This executive is responsible for processing all flight systems.
--|    Processing involves handling all connections between the flight
--|    systems and processing each system.
--|
```

```
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*************************************************************

with Flight_System_Names;
with Flight_Executive_Connection_Manager;

with Engine_System;

package body Flight_Executive is

   type Active_In_Frame is
     array (Flight_System_Names.Name_Of_A_Flight_System) of Boolean;

   Its_Time_To_Do : constant array (Global_Types.Execution_Sequence) of
                       Active_In_Frame :=
     (Global_Types.Frame_1_Modules_Are_Executed =>
        (Flight_System_Names.Engine => (True), others => (False)),

      Global_Types.Frame_2_Modules_Are_Executed =>
        (Flight_System_Names.Electrical => (True), others => (False)),

      Global_Types.Frame_3_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_4_Modules_Are_Executed => (others => (False)),

      Global_Types.Frame_5_Modules_Are_Executed =>
        (Flight_System_Names.Engine => (True), others => (False)),

      Global_Types.Frame_6_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_7_Modules_Are_Executed => (others => (False)),
      Global_Types.Frame_8_Modules_Are_Executed => (others => (False)));



   procedure Update_Flight_Executive (Frame : in
                                      Global_Types.Execution_Sequence) is
--|*************************************************************
--| Description:
--|    flight executive. Performs process connections and update
--|    as an atomic action for each system.
--|
--| Parameter Description:
--|    frame is the current executing frame
--|
--| Notes:
```

```
--|    none
--|*****************************************************************

  begin
    for A_System in Flight_System_Names.Name_Of_A_Flight_System loop
      if Its_Time_To_Do (Frame) (A_System) then
        case A_System is

          when Flight_System_Names.Electrical =>
            null;

          when Flight_System_Names.Engine =>
            Flight_Executive_Connection_Manager.
              Process_External_Connections_To_Engine_System;
            Engine_System.Update_Engine_System;
        end case;
      end if;
    end loop;

  end Update_Flight_Executive;


begin      -- flight_Executive
  null;

end Flight_Executive;
```

C.14. Package Flight_System_Names

```
--|*****************************************************************
--| Module Name:
--|    Flight System Names
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Names all systems under flight executive
--|---------------------------------------------------------------
--| Module Description:
--|    Provides the names of all systems under flight executive.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|---------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|---------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
```

```
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

package Flight_System_Names is

   type Name_Of_A_Flight_System is (Electrical, Engine);

   type Aircraft_Engines is (Engine_1, Engine_2, Engine_3, Engine_4);

end Flight_System_Names;
```

C.15. Package Flight_Executive_Connection_Manager

```
--|*****************************************************************
--| Module Name:
--|    Flight Executive Connection Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Describes and processes all connections between flight systems
--|-----------------------------------------------------------------
--| Module Description:
--|    This package is responsible for proccessing all connections between
--|    systems at all levels within the Flight Executive.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

package Flight_Executive_Connection_Manager is

   procedure Process_External_Connections_To_Engine_System;
--|*****************************************************************
```

```
--| Description:
--|    This procedure processes all connections between the engine
--|    system and the other systems at the flight executive level.
--|    Processing of connections means to make the system consistent
--|    with its environment.
--|
--| Parameter Description:
--|    none
--|*********************************************************************

end Flight_Executive_Connection_Manager;
```

C.16. Package body Flight_Executive_Connection_Manager

```
--|*********************************************************************
--| Module Name:
--|    Flight Systems Connection Manager
--|
--| Module Type:
--|    Package Body
--|
--|---------------------------------------------------------------------
--| Module Description:
--|    The procedure below defines all connections for passing data
--|    between flight systems.  Each connection is handled by a procedure
--|    call.
--|
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|---------------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|---------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*********************************************************************

package body Flight_Executive_Connection_Manager is

   procedure Process_External_Connections_To_Engine_System is separate ;
--|*********************************************************************
--| Description:
--|    This procedure processes all connections between the engine
--|    system and the other systems at the flight executive level.
--|    Processing of connections means to make the system consistent
--|    with its environment.
--|
```

```
--| Parameter Description:
--|    none
--|
--| Notes:
--|    none
_|*********************************************************************

end Flight_Executive_Connection_Manager;
```

C.17. Separate Procedure body Process_External_Connections_To_Engine_System

```
--|*********************************************************************
--| Module Name:
--|    Process_External_Connections_To_Engine_System
--|
--| Module Type:
--|    Separate Procedure Body
--|
--| Module Purpose:
--|    Process connections between an engine system and all external
--|    systems.
--|---------------------------------------------------------------------
--| Module Description:
--|    This procedure processes all connections between an engine system
--|    and external systems. Processing of connections means to make
--|    the system consistent with its environment.
--|
--| Parameter Description:
--|    none
--|
--| References:
--|    Design Documents:
--|    none
--|
--|    User's Manual:
--|    none
--|
--|    Testing and Validation:
--|    none
--|
--| Notes:
--|    none
--|---------------------------------------------------------------------
--| Modification History:
--|    25Aug87   kl   created
--|
--|---------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*********************************************************************

with Standard_Engineering_Types;
with Flight_System_Names;

with Burner_Object_Manager;
```

```
with Engine_System_Aggregate;
with Rotor2_Object_Manager;

separate (Flight_Executive_Connection_Manager)

procedure Process_External_Connections_To_Engine_System is

   Integrated_Drive_Energy : Integer;

--
-- A local variable is defined to store the value spark when it is read from
-- the ignition system.  This is a convention, described in the SEI Ada
-- Coding Guidelines, to restrict the spread of embedded function calls, i.e.
-- function calls as parameters within other function calls.
--
   subtype Spark_Type is Integer range 0 .. 20;
   Some_Spark : Spark_Type;
   The_Burner_Spark : Burner_Object_Manager.Spark;

   function Spark_Conversion (In_Spark : in Spark_Type)
                    return Burner_Object_Manager.Spark is
      --|*********************************************************************
      --| Description:
      --|    This function performs a type conversion.  It converts
      --|    the spark from the Ignition to a spark that the
      --|    Burner_Object_Manager can accept.  This is done
      --|    as an example of how the type conversions can be used to
      --|    connect objects which either communicate through a
      --|    valve/regulator, or need different grains of coarseness of
      --|    the information.
      --|    In this case we are assuming that the Ingition system
      --|    needs finer information about the spark than the Burner.
      --|
      --| Parameter Description:
      --|    In_Spark is the spark that the Ignition supplies.
      --|    return Spark is the spark returned for the Burner
      --|*********************************************************************
   begin
      case In_Spark is
         when 0 .. 2 =>
            RETURN Burner_Object_Manager.None;
         when 3 .. 9 =>
            RETURN Burner_Object_Manager.Low;
         when 10 .. 20 =>
            RETURN Burner_Object_Manager.High;
      end case;
   end Spark_Conversion;

begin  -- Process_Engine_Connections_To
   --
   -- All engine external connections are handled in this procedure.
   -- Each engine has the same kind of connections, but each engine is
   -- connected to different instances of other objects.  Thus all engines
   -- are handled alike here.  The different connections are described by
   -- the engine_system_aggregate package.
   --
   for An_Engine in Flight_System_Names.Aircraft_Engines loop
      --
      -- Get_Air_From (the_environment);
      -- Give_Air_To (a_diffuser);
      --    goes here
      --


      --
      -- Get_Mach_Number_From (the_airframe);
      -- Give_Mach_Number_To (a_diffuser);
```

```
      --     goes here
      --

      --
      -- Get_Discharge_Pressure_From (the_cabin_air);
      -- Get_Discharge_Pressure_From (the_air_conditioning_system);
      --     any processing of these two pieces of information goes here
      -- Give_Discharge_Pressure_To (a_bleed_valve);
      --     goes here
      --

      --
      -- Get_Torque_From (the_hydraulic_system);
      -- Get_Torque_From (the_oil_system);
      -- Get_Torque_From (the_starter_system);
      -- Get_Torque_From (the_fuel_system);
      -- Get_Torque_From (the_electrical_system);
      --     any processing of these five pieces of information goes here
      -- Give_Torque_To (a_rotor2);  --     goes here
      --
      -- for now we are just showing one of these five connections, the one
      -- from the electrical system.  for the complete system, all five pieces
      -- of information would be gathered and processed before passing the
      -- information to the Rotor2.
      --
      Integrated_Drive_Energy := 15;

      Rotor2_Object_Manager.Give_Torque_To
        (A_Rotor2 => Engine_System_Aggregate.Engines (An_Engine).The_Rotor2,
         The_Torque => Standard_Engineering_Types.Torque
                       (Integrated_Drive_Energy));

      --
      -- Get_Fuel_Flow_From (the_fuel_system);
      -- Give_Fuel_Flow_To (a_burner);
      --     goes here
      --

      --
      -- get Spark from the Ignition and feed it to the Engine Burner.
      --
      Some_Spark := 15;

      The_Burner_Spark := Spark_Conversion (Some_Spark);

      Burner_Object_Manager.Give_Spark_To
        (A_Burner => Engine_System_Aggregate.Engines (An_Engine).The_Burner,
         Given_Spark => The_Burner_Spark);
      --
    end loop ;
end  Process_External_Connections_To_Engine_System;
```

C.18. Package Engine_System

```
--|*****************************************************************
--| Module Name:
--|    Engine_System
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package contains the single procedure call to update the
```

```
--| simulation of an Engine System.  It is the sole interface to the Engines
--| from the perspective of the executive.
--|-------------------------------------------------------------
--| Module Description:
--|    The single operation provided by this package is parameterized with
--| the name of the engine to be updated.  The operation accomplishes
--| two sets of lower-level operations:
--|   -one to update the state of the objects at the boundries of the
--|    engine system which have connections (interfaces) with objects
--|    in other systems external to the engine system.
--|   -and another to update all objects internal to the engine system
--|    based on the connections (interfaces) between each other.
--| Specifying the name of the engine allows the work to be spead out
--| across the available processing time, and pushes this decision up
--| to a higher, more intelligent being (the executive) to choose the
--| order of updating the engines in the engine system.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-------------------------------------------------------------
--| Modification History:
--| 21AUG87    CPP Creation
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*************************************************************

package Engine_System is

   procedure Update_Engine_System;
   --|*********************************************************************
   --| Description:
   --|    Allows the simulation of the Engine System to be updated
   --|    and made consistent.  Then other systems dependent upon
   --|    the Engine System can access the consistent state of the
   --|    Engine System.  It is an atomic action.
   --|
   --| Parameter Description:
   --|    Given_Engine_Name
   --|    It's type is declared in Flight_System_Names and is used
   --|    to allow a higher, more intelligent being (the executive) to
   --|    choose the order of updating the engines in the
   --|    engine system.
   --|
   --|*********************************************************************

end Engine_System;
```

C.19. Package body Engine_System

```
--|*****************************************************************
--| Module Name:
--|    Engine_System
--|
--| Module Type:
--|    Package Body
--|
--|-----------------------------------------------------------------
--| Module Description:
--|    The operation provided by this package allows the "user" to
--| update the state of an engine, i.e. update the state of the
--| objects which simulate the individual parts of the engine.
--|
--| Because this system is at the level above the object
--| managers, we have decided that the system will internally
--| implement the conection manager at this level since we don't
--| have to go around and touch the object managers and tell them
--| to update themselves.  The object managers update themselves
--| (thier state) when the connection is made and the state
--| information is given to them.
--|
--|
--|
--| References:
--|    Design Documents:
--|    Engine Object Diagram.
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--| THIS IS NOT A FULL IMPLEMENTATION!!!
--| The code is done to demonstrate the process of connecting objects
--| in a system.
--|
--|    The connection manager wasn't implemented at this level
--| for the reasons stated above in the Module Description.
--|
--| Once the Engine system has been updated, i.e., it's state
--| made consistent, any objects whose state is needed by objects
--| in other systems can be had by directly accessing the object
--| and getting it's state.
--|
--| All internal routines preform a type conversion on the data
--| when the data is transfered from engine object to engine object.
--| This is done to allow flexibility and greater potential for reuse
--| of object managers.  Another reason for type conversions, which
--| is related to the flexibility issue, is that there may be something
--| to model at the connection between the objects, i.e. a valve,
--| regulator, etc., for which an object manager is not necessary.
--| Therefore, any calculations or transformations which need to occur
--| and be modelled at the connection can be made when the connection
--| between the objects occur.
--|
--|-----------------------------------------------------------------
--| Modification History:
--|    24AUG87    cpp    Creation
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    Distribution unlimited
--|
--|    No warranty is implied.
```

```
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense."
--|
--|*****************************************************************

with Standard_Engineering_Types;
with Flight_System_Names;

with Engine_System_Aggregate;
with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;

package body Engine_System is

   procedure Update_Engine_System is
      --|*************************************************************
      --| Description:
      --|    Allows the simulation of the Engine System to be updated
      --|    and made consistent.  Then other systems dependent upon
      --|    the Engine System can access the consistent state of the
      --|    Engine System.  It is an atomic action.
      --|
      --|    The object managers which simulate the various parts of the
      --|    engine, thus comprising the engine system, are
      --|    needed to update the system's state are the following:
      --|    Diffuser_Object_Manager
      --|    Rotor1_Object_Manager
      --|    Fan_Duct_Object_Manager
      --|    Rotor2_Object_Manager
      --|    Burner_Object_Manager
      --|    Exhaust_Object_Manager
      --|    Engine_Casing_Object_Manager
      --|    The connections between these objects and the state information
      --|    flowing between the objects were derived solely from the
      --|    Engine Physical Model Diagram in ???.
      --|
      --| Parameter Description:
      --|    Given_Engine_Name
      --|    It's type is declared in Engine_Names and is used to allow
      --|    a higher, more intelligent being (the executive) to
      --|    choose the order of updating the engines in the
      --|    engine system.
      --|
      --| Note:
      --|    This routine models the connection manager for this level.
      --|*************************************************************

      Diffuser_Discharge_Pressure : Standard_Engineering_Types.Pressure;
      Diffuser_Discharge_Temperature : Standard_Engineering_Types.Temperature;
      Diffuser_Discharge_Air_Flow : Standard_Engineering_Types.Air_Flow;

   begin
      for An_Engine in Flight_System_Names.Aircraft_Engines loop
         --
         -- Model the connection characterized by the dependence of the
         -- Engine Casing on the Diffuser for Pneumatic_Energy.
         --
         Diffuser_Object_Manager.Get_Discharge_Air_From
           (A_Diffuser =>
              Engine_System_Aggregate.Engines (An_Engine).The_Diffuser,
            Returning_Discharge_Pressure => Diffuser_Discharge_Pressure,
            Returning_Discharge_Temperature =>
              Diffuser_Discharge_Temperature,
            Returning_Discharge_Air_Flow => Diffuser_Discharge_Air_Flow);
```

```
        Engine_Casing_Object_Manager.Give_Inlet_Air_Flow_To
          (A_Engine_Casing => Engine_System_Aggregate.Engines (An_Engine).
                              The_Engine_Casing,
           Given_Inlet_Air_Flow => Diffuser_Discharge_Air_Flow);
        --
        -- .
        -- .
        -- .
        --

    end loop ;
  end Update_Engine_System;

end Engine_System;
```

C.20. Package Engine_System_Aggregate

```
--|*********************************************************************
--| Module Name:
--|    Engine_System_Aggregate
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    This package names the TurboRotor1 Engines and their parts.
--|---------------------------------------------------------------------
--| Module Description:
--| A TurboRotor1 Engine is an aggregate of parts:
--|    Diffuser,
--|    Rotor1,
--|    Fan_Duct,
--|    Rotor2,
--|    Bleed_Valve,
--|    Burner,
--|    Exhaust,
--|    Engine_Casing.
--|
--| The parts of a TurboFan1 Engine are objects which have state
--| and suffer actions. Each part is managed by it's own object
--| manager. This package builds the four engines by calling on
--| the various object managers to create the parts. It then stores
--| references to the parts in a constant array indexed by the
--| Aircraft_Engines. The constant array
--| is created when the package is elaborated. The constant array is
--| called Engines. A part of an Engine is referenced as:
--|    Engines(Engine_Name).The_<part_kind>
--| For example, the Rotor1 of the second Engine is:
--|    Engines(Engine_2).The_Rotor1
--|
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--| Optimizations which were implemented: the initialization of Engines
```

```
--| occurs at the declaration of the Object instead of the body because
--| the number of engines and the parts shouldn't change, thus the object
--| was also made to be constant array of Engines.
--|
--|-------------------------------------------------------------
--| Modification History:
--|   20APR87   CPP   Creation
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   Distribution unlimited
--|
--|   No warranty is implied.
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense."
--|
--|*************************************************************
with Flight_System_Names;

with Bleed_Valve_Object_Manager;
with Burner_Object_Manager;
with Diffuser_Object_Manager;
with Engine_Casing_Object_Manager;
with Exhaust_Object_Manager;
with Fan_Duct_Object_Manager;
with Rotor1_Object_Manager;
with Rotor2_Object_Manager;

package Engine_System_Aggregate is

  type Engine_Representation is
    record
      -- Defines an engine representation as consisting of:
      The_Diffuser : Diffuser_Object_Manager.Diffuser;
      The_Rotor1 : Rotor1_Object_Manager.Rotor1;
      The_Fan_Duct : Fan_Duct_Object_Manager.Fan_Duct;
      The_Rotor2 : Rotor2_Object_Manager.Rotor2;
      The_Bleed_Valve : Bleed_Valve_Object_Manager.Bleed_Valve;
      The_Burner : Burner_Object_Manager.Burner;
      The_Exhaust : Exhaust_Object_Manager.Exhaust;
      The_Engine_Casing : Engine_Casing_Object_Manager.Engine_Casing;
    end record ;

  Engines : constant array (Flight_System_Names.Aircraft_Engines) of
              Engine_Representation :=
    -- Defines a constant array which holds all 4 engines in the system
    -- and initializes them (i.e. all their parts)
    (Flight_System_Names.Engine_1 =>
      (The_Diffuser => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1 => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2 => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner => Burner_Object_Manager.New_Burner,
       The_Exhaust => Exhaust_Object_Manager.New_Exhaust,
       The_Engine_Casing =>
         Engine_Casing_Object_Manager.New_Engine_Casing),

     Flight_System_Names.Engine_2 =>
      (The_Diffuser => Diffuser_Object_Manager.New_Diffuser,
       The_Rotor1 => Rotor1_Object_Manager.New_Rotor1,
       The_Fan_Duct => Fan_Duct_Object_Manager.New_Fan_Duct,
       The_Rotor2 => Rotor2_Object_Manager.New_Rotor2,
       The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
       The_Burner => Burner_Object_Manager.New_Burner,
```

```
        The_Exhaust => Exhaust_Object_Manager.New_Exhaust,
        The_Engine_Casing =>
          Engine_Casing_Object_Manager.New_Engine_Casing),

      Flight_System_Names.Engine_3 =>
        (The_Diffuser => Diffuser_Object_Manager.New_Diffuser,
         The_Rotor1 => Rotor1_Object_Manager.New_Rotor1,
         The_Fan_Duct => Fan_Duct_Object_Manager.New_Fan_Duct,
         The_Rotor2 => Rotor2_Object_Manager.New_Rotor2,
         The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
         The_Burner => Burner_Object_Manager.New_Burner,
         The_Exhaust => Exhaust_Object_Manager.New_Exhaust,
         The_Engine_Casing =>
           Engine_Casing_Object_Manager.New_Engine_Casing),

      Flight_System_Names.Engine_4 =>
        (The_Diffuser => Diffuser_Object_Manager.New_Diffuser,
         The_Rotor1 => Rotor1_Object_Manager.New_Rotor1,
         The_Fan_Duct => Fan_Duct_Object_Manager.New_Fan_Duct,
         The_Rotor2 => Rotor2_Object_Manager.New_Rotor2,
         The_Bleed_Valve => Bleed_Valve_Object_Manager.New_Bleed_Valve,
         The_Burner => Burner_Object_Manager.New_Burner,
         The_Exhaust => Exhaust_Object_Manager.New_Exhaust,
         The_Engine_Casing =>
           Engine_Casing_Object_Manager.New_Engine_Casing));

end Engine_System_Aggregate;
```

References

[1] Booch, Grady.
Software Engineering with Ada.
The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[2] Booch, Grady.
Software Components with Ada.
The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[3] Hesse, Walter J. and Mumford, Nicholas V. S., Jr.
Jet Propulsion for Aerospace Applications.
Pitman Publishing Corporation, New York, NY, 1964.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-30	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88-31
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) AN OOD PARADIGM FOR FLIGHT SIMULATORS, 2nd EDITION	
12. PERSONAL AUTHOR(S) Kenneth J. Lee, Michael S. Rissman, Richard D'Ippolito, Charles Plinta, Roger Van Scoy	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) September 1988 15. PAGE COUNT 120
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) object-oriented software engineering Ada flight simulators

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents a paradigm for object-oriented implementations of flight simulators. It is a result of work on the Ada Simulator Validation Program (ASV) carried out by members of the technical staff at the Software Engineering Institute (SEI).

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☑ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE